An Architectural Framework for the Design, Analysis and Implementation of Interactive Systems

Alexandre R.J. François, *Member, IEEE*

A. R.J. François is a 2007-8 Fellow of the Radcliffe Institute for Advanced Study at Harvard University, and a research assistant professor of computer science in the Viterbi School of Engineering at the University of Southern California.

Abstract

This article presents the Software Architecture for Immersipresence (SAI) framework for the design, analysis and implementation of interactive software systems. SAI defines a formal architectural style whose data and processing models capture temporal properties of computational primitives. SAI's asynchronous concurrent processing model allows designing for optimal (theoretical) system latency and throughput. The modularity and scalability of the style facilitate distributed code development, testing, and reuse, as well as fast system design and integration, maintenance and evolution. A graph-based notation for architectural designs affords intuitive and scalable system representation. The formalism allows a range of intermediate-level representations from conceptual to logical level. SAI favors the encoding of system logic in the structural organization of simple computing components, rather than in computationally complex individual components. Designs exhibit a rich variety of structural and functional architectural patterns, suitable for systematic study and re-use. The open source Modular Flow Scheduling Middleware provides a multi-threaded, cross-platform implementation of SAI's primitives. The SAI/MFSM framework has been used in the design and implementation of numerous interactive systems in research, education and performance settings. This article illustrates the definition and use of SAI with examples from computer vision.

Index Terms

Software Architecture; Interactive Systems.

I. INTRODUCTION

This article presents a detailed and complete description of the Software Architecture for Immersipresence (SAI) framework for the design, analysis and implementation of interactive software systems. The Integrated Media Systems Center, in the late 1990's [1], coined the term *immersipresence* to denote the engineering of naturalistic remote immersive interaction, *raison d'être* of the center. Immersipresence draws on many traditional fields of engineering, including signal processing, networking, database systems, computer vision, computer graphics, haptics, to name a few. In this context, the SAI project addresses the engineering of software that integrates algorithmic and hardware solutions from disparate fields into working systems that operate under the physical constraints imposed by their interactive nature. These constraints include (soft) real-time performance, low latency and consistent synchronization.

A. Synopsis

The SAI framework [2] aims to facilitate the design and implementation of interactive software systems. SAI defines a formal architectural style [3] whose underlying extensible data model and hybrid (shared memory and message-passing) distributed asynchronous concurrent processing model allow natural and efficient manipulation of data streams. The modularity and scalability of the style facilitate distributed code development, testing, and reuse, as well as fast system design and integration, maintenance and evolution.

Time and timing are critical concepts in all interactive applications. Interacting agents perceive data as streams of dynamic information that, in order to make sense, must respect synchronization constraints within streams (temporal precedence) and across streams (precedence and simultaneity). A reflection on the temporal nature and properties of computational elements shaped the definition of SAI's primitives and principles. SAI's concurrent and asynchronous processing model ensures that architectural designs specify costly synchronization structures explicitly, only when needed. In particular, such designs can be optimal with respect to latency and throughput simultaneously, where sequential models sacrifice latency for throughput (and vice-versa), and pipelined concurrent models may force suboptimal latency. SAI's data model introduces the distinction between volatile and persistent data. Volatile data flow on streams and remain in a system for a limited fraction of its lifetime. Examples include time samples of a signal (dense stream) and events (sparse stream). Persistent data remain in the system for the whole duration of its lifespan (or a significant fraction thereof) and are held in shared repositories. Examples include process parameters.

Designing and integrating inter-disciplinary interactive systems is as much a human communication challenge as an engineering one. SAI is designed to be used by, and useful to, teams of people with diverse degrees of technical expertise (and interest). A graph-based notation for architectural designs affords intuitive and scalable system representation at the conceptual and logical levels. SAI allows a continuum of intermediate-level representations from conceptual to logical level. SAI promotes the encoding of system logic in the structural organization of simple computing components, rather than in the complexity of the computations carried out by individual components. SAI designs exhibit a rich variety of structural and functional *architectural patterns*, suitable for systematic study and re-use.

An open source architectural middleware, the Modular Flow Scheduling Framework (MFSM) [4]. provides cross-platform code implementation of SAI's architectural abstractions. The MFSM project also comprises extensive documentation, including user guide, reference guide and tutorials, all available on the project Web site.

B. Applications

The SAI framework has been used in the design and implementation of numerous interactive systems. In the computer vision domain, early interactive systems designed and developed with the SAI framework include *Video Painting* [5], a real-time mosaicing system, and *Virtual Mirror* [5] [6] [7], a handheld mirror simulation. Other interactive systems are described in [2]. An early demonstration that performed live video mapping on an animated 3D model [8] illustrated the design of asynchronous concurrent component for interactive computer graphics. As SAI and associated tools became more mature, they allowed more ambitious cross-disciplinary projects. The SAI framework has facilitated the collaborative design and successful implementation of original and innovative interactive music systems [9]. Representative projects include *Music on the Spiral Array . Real-Time* (MuSA.RT) [10] [11], an interactive music analysis and visualization system; the *Expression Synthesis Project* (ESP) [12] [13] [14], a driving interface for generating expressive musical performances; and, *Multimodal Interaction for Musical Improvisation* (MIMI) [15], an interactive musical improvisation system that provides visual feedback to the performer.

The SAI framework was instrumental in the successful implementation of graduate and undergraduate courses that revolve around a class-wide term project. A graduate level seminar course, titled *Integrated Media Systems*, taught in Fall 2002 at the University of Southern California (USC), introduced students to the specific technological and organizational difficulties of designing and building interactive and media-rich integrated systems [2] [16] [17]. Using SAI and MFSM, through a supervised, systematic, iterative and modular process, 24 students developed a playable on-line 3D soccer game in only two months. None of the students had used SAI before, and none had ever participated in such a project. The design of a special session of the undergraduate course *Principles of Software Development* [18] (the last of a series of four required programming courses at USC) for students enrolled in, or interested in pursuing, USC's new computer science Games major, adopted a similar class-wide collaborative

project. In the Spring 2008 session, 8 students, using the SAI/MFSM framework, collaboratively designed and implemented *Crosswinds* [19], a networked video adaptation of an original student-designed board game. The SAI formalism allows high level understanding of the overall system organization, precise composition rules, and strong structure for code development, a combination that was, in both cases, essential in the successful completion of a class-wide collaborative project. The class project in these courses represented a major innovation compared to traditional class projects, in which individuals or small teams develop small-scale independent or identical projects.

This article calls on a particular application, visual tracking, to help illustrate and ground the concepts and principles that drive the definition and use of the SAI framework. A simple face tracking demonstration serves as a running example during the description of the SAI style, and ties into the design case study of a more complex vision system. Stevi, a high-level vision system for a personal service robot, shows the potential of the SAI approach for building ambitious systems that integrate with other software and hardware components in interactive applications.

C. Outline

The remainder of this article is organized as follows. Section II gives a short introduction to the visual tracking examples. Section III presents a formal definition of the SAI architectural style. Section IV gives an overview of the MFSM architectural middleware. Section V illustrates architectural design through refinement of simple visual tracking demonstration system. Section VI illustrates architectural design through specialization and composition of architectural patterns in the design of Stevi, a high-level vision system for a personal service robot. Section VII compares and contrasts SAI with representative related efforts in software architecture and other fields of application. Section VIII summarizes the main features and properties of the SAI architectural framework and offers directions for future research.

II. Visual Tracking with CAMSHIFT

This section gives a succinct introduction to the visual tracking examples that will help ground SAI concepts into concrete situations, and illustrate the design methodology afforded by the framework.

A. CAMSHIFT Tracking

The Continuously Adaptive Mean SHIFT (CAMSHIFT) algorithm [20] detects and tracks a pattern, for example a person's face, in a video stream.

The "plain" mean shift algorithm [21] is a general purpose, robust, non-parametric, iterative gradient descent algorithm for finding the mode (density maximum) of a probability distribution. For detecting, in a still color picture, an instance of a target visually characterized by a color model (histogram), a mean shift-based method first computes a probability distribution from the image by attributing to each color pixel its value in the target's histogram. Starting from a reasonable position in the image, the iterative gradient descent optimally converges towards the location of a model instance. The only parameter to the algorithm, the size of the area to use for sampling the distribution in the gradient descent, is a fixed parameter; construction of the color model and choice for a good starting position are out of the scope of the algorithm *per se*. The mean shift algorithm exhibits, under reasonable conditions, very attractive convergence and robustness properties, and has proved quite efficient in practice.

Mean shift-based color tracking [22] applies the same method to successive frames of a video stream to detect and track instances of a color model. For a given target, due to temporal continuity across adjacent frames, starting the gradient descent process from the position where the instance was detected in the previous frame, may result in possibly large computational savings.

CAMSHIFT extends mean shift-based color tracking by continuously computing the size and orientation of the tracked area, which is fed back to the mean shift process as the size of the sampling area in the next frame. Figure 1 shows the position, size and orientation of the mode as detected in the probability distribution (left) and overlaid on the input frame (right).

B. CAMSHIFT in OpenCV

The Intel OpenCV Library [23] [24] features an object-oriented implementation of CAMSHIFT, demonstrated in a real-time head tracking application. The area of application specifically considered is that of Perceptual User Interfaces [25], in this particular case using head movements as seen in a face shot video stream to control various interactive programs. The color model used for the head is actually a skin color tone model, initialized by sampling an area specified manually in one image.

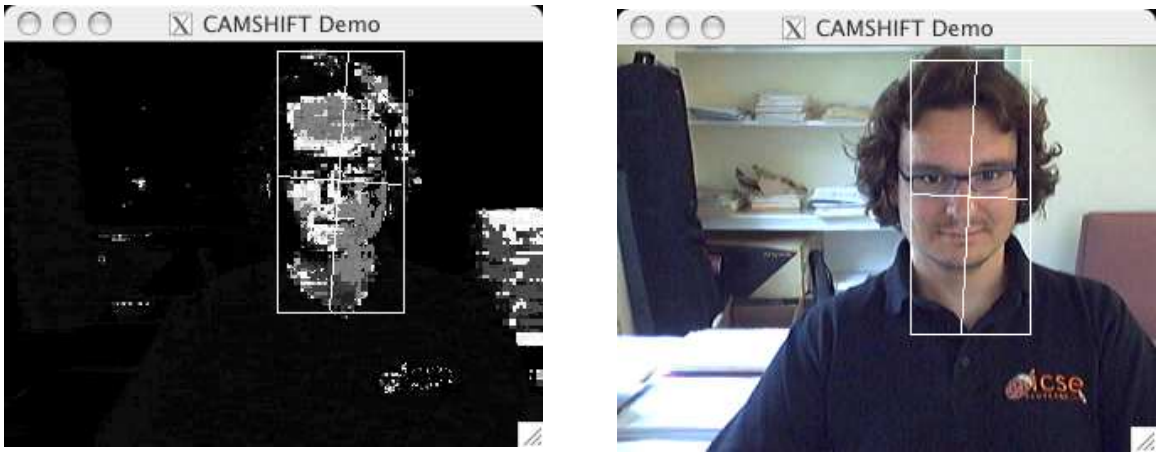

Fig. 1. CAMSHIFT-based face tracking. Left: mode detection in skin color model probability image. Right: inferred face area.

1) Code architecture: OpenCV implements individual features useful in developing a CAMSHIFT-based tracking system. The code adopts an object-oriented organization structure. Low-level features include general purpose data structures (e.g. histograms) and functions (e.g. color conversion, back projection); intermediate-level features include more specialized functions that package calls to general purpose functions in specific algorithms such as mean shift or CAMSHIFT; at the highest level, a black-box class encapsulates all necessary data structures and hides all intermediate library calls internally.

2) System architecture: The CAMSHIFT demonstration that shipped with OpenCV version beta 3.1 serves as a baseline for the examples developed in this article. The demonstration software is designed and implemented using Microsoft's DirectShow [26] pipes-and-filters architecture library. Figure 2 shows the corresponding graph in DirectShow's GraphEdit tool. The architecture of the software system reflects the highest level code architecture: CAMSHIFT is encapsulated in a single filter that, for demonstrations purposes, consumes images and produces images.

Through the remainder of the article, examples and case studies reproduce this simple demonstration system using the SAI formalism, refine the design, open the system's architecture, make explicit its core architectural pattern, namely a feedback loop. A case study demonstrates the integration of a generalized version of this core architectural pattern with other important patterns in a more complex vision system.

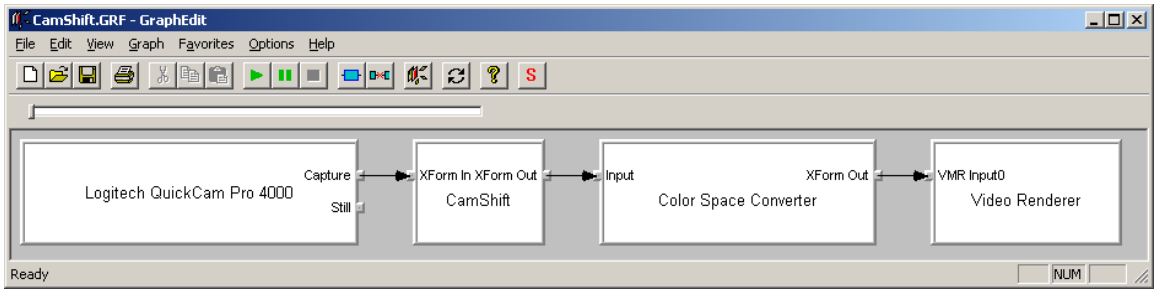

Fig. 2. DirectShow graph for the OpenCV CAMSHIFT tracker demonstration (Intel OpenCV version beta 3.1).

III. The SAI Architectural Style

This section presents a formal definition of the SAI architectural style, in terms of components, connectors and constraints [27]. Graphical symbols are introduced to represent each element type. Together these symbols constitute a graph-based notation system for representing architectural designs. In addition, when color is available, the following color coding will be used: green for processing, red for persistent data, blue for volatile data. Figure 3 presents an overview of the SAI primitives in their standard notation.

A. Components, Connectors and Constraints

The SAI style defines two types of components: *cells* and *repositories*[1]. Cells are processing centers. They do not store any state data related to the specific computations they carry. The cells constitute an extensible set of specialized components that implement specific algorithms. Each specialized cell type is identified by a type name (string), and is logically defined by its input data, its parameters and its output. Cell instances are represented graphically as green squares. A cell can be active or inactive, in which case it is transparent to the system. Repositories hold shared persistent data. Repository instances are represented as red disks or circles. Two types of connectors link cells to cells and cells to repositories. Cell to repository connectors give the cell access to the shared data held in the repository. Cell to cell connectors define data conduits for volatile data flows, or *streams*, and specify cell process dependency. Note that the semantics of these connectors are explicitly different from that of the connectors in dataflow networks,

[1]Repositories were named *sources* in early descriptions of the SAI style.

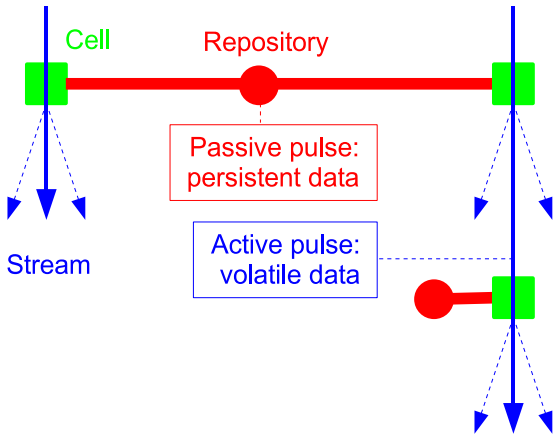

Fig. 3. Overview of the SAI primitives. Cells are represented as squares, repositories as circles. Source-cell connections are drawn as fat lines, while cell-cell connections are drawn as thin arrows crossing over the cells. When color is available, cells are colored in green (reserved for processing); repositories, repository-cell connections, passive pulses are in colored in red (persistent information); streams and active pulses are colored in blue (volatile information).

that are by definition First-In-First-Out channels. In SAI, stream connectors do not convey any constraint on the time ordering of the data flowing through them.

Cell and repository instances interact according to the following rules. A cell must be connected to exactly one repository, which holds its persistent state data. A repository can be connected to an arbitrary number of cell, all of which have concurrent shared memory access to the data held by the source. A repository may hold data relevant to one or more of its connected cells, and should hold all the relevant data for each of its connected cells (possibly with some overlap). Cell-repository connectors are drawn as either double or fat red lines. They may be drawn across cells (as if cells were connected together by these links) for graphical layout convenience. A cell can be connected to exactly one upstream cell, and to an arbitrary number of downstream cells. Streams (and thus cell-cell connections) are drawn as thin blue arrows crossing over the cells.

1) Data Model: Data, whether persistent or volatile, is held in *pulses*. Each repository contains a *passive pulse*, which encodes the instantaneous state of the persistent data structures held by the repository. Volatile data flows in streams, that are temporally quantized into *active pulses*. An active pulse carries all the causally related data in a stream, corresponding to a same original

time stamp. Information in a pulse is organized as a mono-rooted composition hierarchy of *node* objects. The nodes constitute an extensible set of atomic data units that implement or encapsulate specific data structures. Each specialized node type is identified by a type name (string). Node instances are identified by a name. The notation adopted to represent node instances and hierarchies of node instances makes use of nested parentheses, e.g.: (NODE_TYPE_ID "Node name" (...) ...). This notation may be used to specify a cell's output (see Table I for an example), and for logical level specification of active and passive pulses. Pulses are represented graphically as a root (solid small disk) and a hierarchy of nodes (small circles); passive pulses may be rooted in the circle or disk representing the repository.

2) Processing Model: Upon entering a cell, an active pulse triggers a series of operations that can lead to the processing of the pulse by the cell (hence the "active" qualifier). Processing in a cell may result in the augmentation of the active pulse (input data), and/or update of the passive pulse (e.g. process parameters). The processing of active pulses is carried concurrently, as the pulses arrive at the cell. In particular, the semantics of the style's primitives do not include pipelining or any other synchronization aspects that would be arbitrary at this level. Such synchronization logic should be explicitly modeled in the form of architectural patterns (sections VI-B and VII-C offer examples of synchronization patterns).

From a practical point of view, since a cell process can only read the existing data in an active pulse, and never modify them (except for adding new nodes), concurrent read access does not require any special precautions. In the case of passive pulses, however, appropriate locking (e.g. through critical sections) must be implemented in the cells' logic to avoid inconsistencies in concurrent shared memory read/write access.

3) Dynamic Data Binding: Passive pulses may hold persistent data relevant to several cells. Therefore, before a cell can be activated, the passive pulse must be searched for the relevant persistent data. As pieces of data are accumulated in active pulses flowing down the streams through cells, it is also necessary for a cell to search each active pulse for its input data. If the specified data structures cannot be found, or if the cell is not active, the pulse is transmitted, as is, to the connected downstream cells. If the input data are found, then the cell process is triggered. When the processing is complete, then the pulse, which now also possibly contains output data nodes produced by the cell, is passed downstream.

Searching a pulse for relevant data, called *filtering*, is an example of run-time data binding.

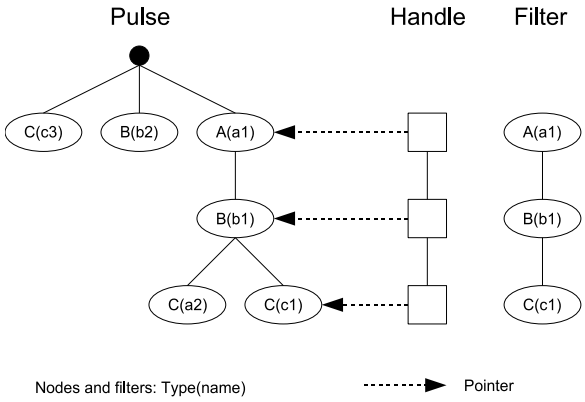

Fig. 4. Pulse filtering. Each cell is associated with its required volatile and persistent data structures, in the form of substructures called active and passive filters (respectively). Pulses are searched for these structures in an operation called filtering, which results in the creation of handles that can be used during processing for direct access to relevant nodes.

The target data fragment is characterized by its structure: node instances types and names and their relationships. The structure is specified as a *filter* or a composition hierarchy of filters. Note that the term filter is used here in its "sifting" sense. Figure 4 illustrates this concept. A filter specifies a node type, a node name or name pattern and eventual subfilters corresponding to subnodes. The filter composition hierarchy is isomorphic to its target node structure. The filtering operation takes as input a pulse and a filter, and, when successful, returns a *handle* or hierarchy of handles isomorphic to the filter structure. Each handle provides a pointer to the node instance target of the corresponding filter. When relevant, optional names inherited from the filters allow to identify individual handles with respect to their original filters.

The notation adopted for specifying filters and hierarchies of filters is nested square brackets. Each filter specifies a node type, a node instance name or name pattern (with wildcard characters), an optional handle name, and an eventual list of subfilters, e.g.: [NODE_TYPE_ID "Node name" *handle_id* [...] ...] (see Table I for an example). Optional filters are indicated by a star, e.g.: [NODE_TYPE_ID "Node name" *handle_id*]*. For any given cell type, the filters' structures and types are fixed. For a given cell instance, the specific filter instance names may change at run-time.

When several targets in a pulse match a filter's name pattern, all corresponding handles are created. This allows the specification of processes whose input (parameters or stream data)

number is not fixed. If the root of the active filter specifies a pattern, the process method is invoked for each handle generated by the filtering (sequentially, in the same thread). If the root of the passive filter specifies a pattern, only one passive handle is generated (pointing to the first encountered node satisfying the pattern).

B. Architectural Design Specification

A particular system architecture is specified at the conceptual level by a set of repository and cell instances, and their inter-connections. Specialized cells may be accompanied by a description of the task they implement. Repository and cell instances may be given names for easy reference. In some cases, important data nodes and outputs may be specified schematically to emphasize some design aspects.

Figure 5 shows the conceptual level architecture in SAI notation for a CAMSHIFT Tracker demonstration system whose structure mirrors that of the demonstration system implemented with DirectShow, described in section II, figure 2. Just as the CAMSHIFT specific portion of the OpenCV CAMSHIFT demonstration is completely encapsulated in a single DirectShow filter, it is encoded here in SAI as a minimal functional unit, composed of a single cell connected to a single repository, which holds a single node, named (conceptually) *CAMSHIFT parameters*, that contains all the persistent information relevant to the processing. Input and output images are volatile. Standard video input and image display modules provide capture and display functionalities.

A logical level description of a design requires to specify, for each cell, its active and passive filters and its output structure, and for each source, the structure of its passive pulse. Table I presents the logical level specification for the CAMSHIFT process cell of figure 5. Filters and nodes are described using the nested square brackets and nested parentheses notations introduced above. By convention, in the cell output specification, (x) represents the pulse's root, (.) represents the node corresponding to the root of the active filter, and (..) represents its parent node.

C. Style Properties

The SAI architectural style shares many of the desirable properties of classical dataflow models. SAI suits intuitive design, emphasizing the flow of data in the system. The modularity and scalability of the model allow distributed development and testing of particular elements,

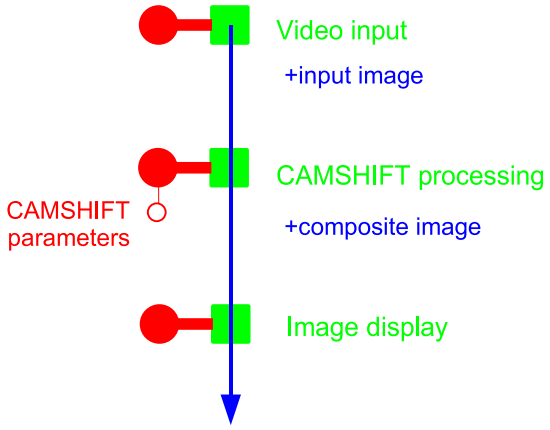

Fig. 5. SAI design for the CAMSHIFT Tracker demonstration: black box design analogous to the original DirectShow version presented in figure 2.

CCamshiftProcessingCell (FsfCCell)	CAMSHIFT_PROCESSING_CELL
Active filter	[IMAGE_NODE "Input frame" *CAMSHIFT_INPUT*]
Passive filter	[CAMSHIFT_PARAMETERS_NODE "CAMSHIFT parameters" *CAMSHIFT_PARAMETERS*]
Output	(x (IMAGE_NODE "CAMSHIFT output"))

TABLE I

A LOGICAL LEVEL DEFINITION FOR THE CAMSHIFT PROCESSING CELL OF FIGURE 5.

and easy maintenance and evolution of existing systems. SAI's primitives naturally afford the design of explicitly concurrent and distributed system. SAI's asynchronous concurrent processing model allows designing for optimal (theoretical) system latency and throughput.

Unlike classical computational approaches, SAI explicitly models volatile and persistent data, and integrates their representation and processing under a single unified formalism. This unification enables explicit and consistent design not only of subsystems that follow one classical paradigm (e.g. data stream or data-centered), but also of the interactions between such subsystems.

An SAI graph presents an intuitive view of the system. Because the visual language in which it is expressed has well defined semantics, the graph is also a formal description. Depending

on the level of detail with which each element is described, the graph may be seen as a conceptual or logical level design. The graph also constitutes a map of the code that implements it (physical level). SAI's principles permit a range of intermediate-level representations from conceptual to physical specifications. SAI promotes the encoding of system logic in the structural organization of simple computing components rather than in the complexity of the computations carried by individual components. SAI designs exhibit a rich variety of structural and functional *architectural patterns*, suitable for systematic study.

The examples developed below in sections V and VI demonstrate how the properties summarized here serve the design of real systems. Other important architectural properties of the SAI style, including natural support for dynamic system evolution, run-time reconfigurability and self monitoring, are topics of ongoing and future research.

IV. Architectural Middleware

This section gives an overview of the Modular Flow Scheduling Middleware (MFSM), an open-source architectural middleware for the SAI style. Developed in C++, MFSM provides cross-platform code support for SAI's architectural abstractions, in the form of an extensible set of classes. A number of software modules regroup specializations that implement specific data structures, algorithms and/or functionalities. They constitute a constantly growing base of open source, reusable code, maintained as part of the MFSM project. The project also comprises extensive documentation, including user guide, reference guide and tutorials, all available on the project web site.

A. MFSM Overview

The code architecture adopted in MFSM introduces a middleware layer as an an abstraction level between low-level services and libraries on the one hand, and software applications on the other hand. The middleware layer comprises of an extensible set of classes that implement software components in the form of SAI style elements. The Flow Scheduling Framework (FSF) library anchors this layer with foundation classes that implement SAI's primitives. The generic extensible data model allows to encapsulate existing data formats and standards as well as low-level service protocols and APIs, and make them available in a system where they can

interoperate. The application layer hosts the software system, specified and implemented as instances of SAI components and their relationships.

In its current implementation, the FSF library contains a set of C++ classes implementing SAI elements: the repository, the nodes and pulses, the cells, the filters, the handles. Nodes and cells are virtual classes that implement the SAI-related behavior of these object types, and from which specialized node and cell types should be derived. The FSF library also contains classes for two implementation related object types: the node and cell factories, and a System object (singleton). An online reference guide provides detailed interface description and implementation notes for all the classes defined in the FSF library.

B. Middleware Properties

The middleware implements all SAI abstractions according to specifications, in a fully multi-threaded (and thread-safe) manner. As a result, systems designed with SAI and implemented with MFSM are automatically multi-threaded, and can directly take advantage of multiple CPUs, hyper-threading and multi-core processor architectures.

The growing collection of functional modules, in the form of both cross-platform and platform-specific code, foster system design space exploration through fast prototyping. Existing modules for generic capabilities allow focusing coding efforts on project-specific components. The MFSM web site lists existing modules (with documentation). Thematic open-source projects regroup documentation, tutorials and code samples illustrating the use of relevant modules in the context of application-specific architectural patterns. For example the Web-Cam Computer Vision (WCCV) project [28] focuses on computer vision algorithms and systems that follow the web cam paradigm: cheap, robust and efficient. In particular, cross-platform code samples illustrate the encapsulation and use of functions and data structures provided by the Intel OpenCV library. The Open Virtuality Engine (OpenVE) project [29] addresses the design and implementation of modular virtuality systems such as video games, and visualization environments. Cross-platform code sample illustrate the partial encapsulation and use of functionalities provided by the OpenGL [30] and OpenGL Utility Toolkit (GLUT) [31] libraries. These pose particularly interesting design and coding challenges, as OpenGL's rendering engine, modeled as a state machine, assumes a single-threaded sequential environment.

Some aspects of the SAI data and processing models, such as filtering, involve non trivial

computations, and the resulting overhead could make the theory impractical. The existence of a significant number of fairly complex interactive systems designed in the SAI style and implemented with MFSM show that, at least for these examples, it is not the case. Experimental scalability tests performed on applications designed in the SAI style, reported in [32] and [5], suggest that: (1) as long as computing resources are available, the overhead introduced by the SAI/MFSM framework remains constant, and (2) the contribution of the different processes are combined linearly. In particular, the middleware does not introduce any significant non-linear complexity in the system. These properties are corroborated by empirical results and experience in developing and operating other systems designed in the SAI style. Theoretical complexity analysis and overhead bounding are topics for future research. This aspect will become especially important in the design and development of hardware- and possibly application-specific middlewares for SAI.

V. Architectural Design: the CAMSHIFT Tracker Application

This section illustrates the scalability of encoding conferred by the SAI primitives. Successive refinements of the architectural design for the CAMSHIFT tracker application make the underlying organization of the system increasingly explicit in its architecture. The expression of operational logic at the architecture level rather than in the code increases the communicability and understandability of the overall system design, while constraining the organization and reducing the complexity of application specific code. The complete study with quantitative coding and performance analyses appears in [16]; the present account focuses on the architectural level.

A. Architectural Refinement

The system design introduced above in Section III-B, and shown in Figure 5, serves as a starting point. Recall that its structure reflects that of the underlying object-oriented code architecture. Figure 6 shows the architectural designs of three functionally equivalent systems. The systems differ by the amount of information about their implementation that is expressed at the architectural level rather than at the code level. Each architectural design constitutes a refinement of the preceding one, and introduces more constraints on its implementation. In the figure, dashed boxes and a “constrain” arrow indicate the relationship between functionally equivalent

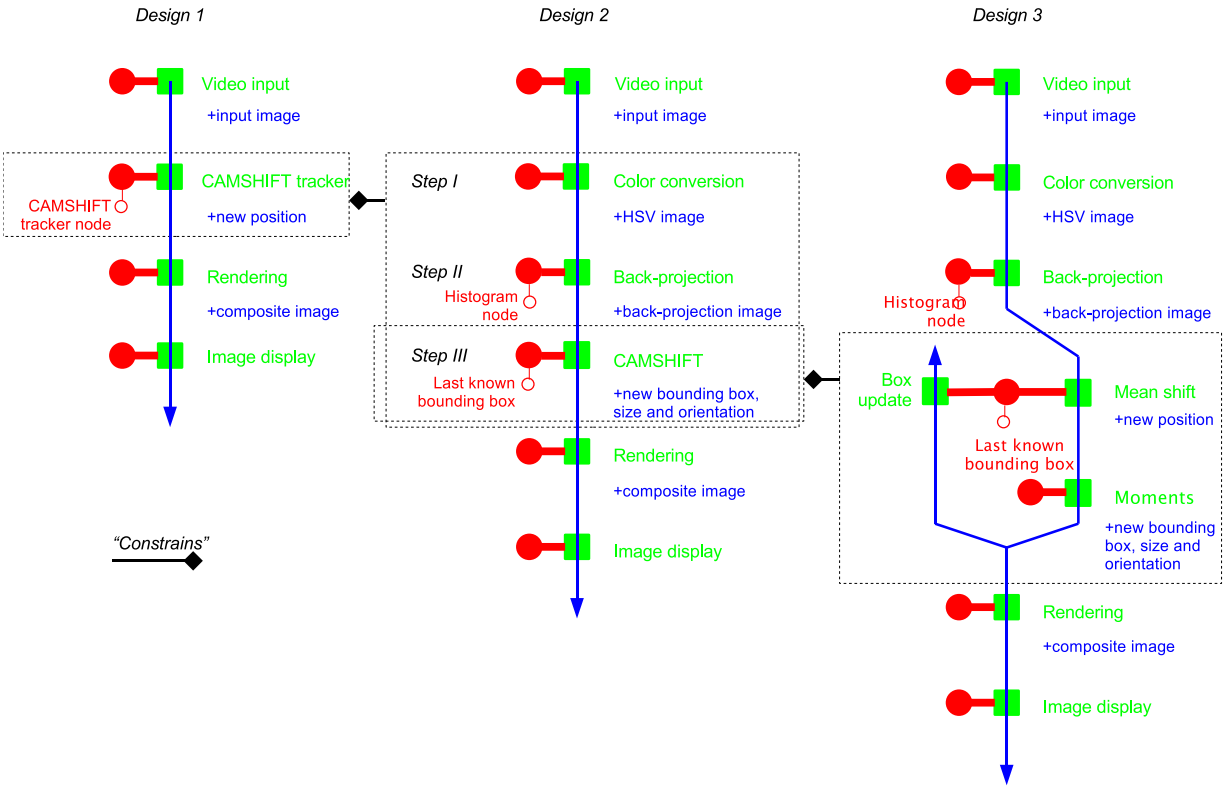

Fig. 6. Increasingly explicit architectural designs for the CAMSHIFT Tracker demonstration. *Design 1*: black-box approach; the dashed line delineates the CAMSHIFT functional unit. *Design 2*: 3-step design; the dashed line delineates the subsystem functionally equivalent to that delineated in *Design 1*. *Design 3*: explicit feedback loop design; the dashed box delineates the subsystem functionally equivalent to the CAMSHIFT unit (step III) in *Design 2*.

architectural elements. The architectural refinement process here parallels the unraveling of layers of encapsulation in the object-oriented code design.

Design 1 separates the rendering of a composite image from the tracking *per se*. This step is consistent with the use of the tracking system beyond a simple demonstration. Separating debugging (or visualization) elements from the production elements being tested or demonstrated facilitates the re-use of functional code in real applications, and allows for more flexibility in the debugging and visualization modalities. The tracking process is modeled as a single cell, with no indication of its internal organization (a "black box" design).

Design 2 reveals three separate, sequential frame processing steps in the tracking process.

1) The color model conversion (here from RGB to HSV) is a generic operation that has no

specificity to the tracking task. Isolating such process facilitates code reuse and simplifies debugging.

2) The computation of the so-called "back projection" image, produces the probability distribution of the given color model (histogram) in the input image data, expressed in the HSV color model. This representation makes explicit the persistent nature of the color model histogram, held in a repository. The corresponding data structure is now clearly (and in style-conformant manner) available to complementary sub-systems that could address the acquisition of the model, which the CAMSHIFT algorithm assumes given. Stevi (section VI-C) offers one possible solution in a more general context.
3) The CAMSHIFT tracking process *per se*, implements the algorithm as described above in Section II-A. Table II shows a logical definition for the CAMSHIFT cell. This design makes explicit the persistent nature of the *Last known bounding box* data structure, whose value initializes the tracking processing in each new frame, and is subsequently updated.

CCamshiftCell (FsfCCell)	CAMSHIFT_CELL
Active filter	[IMAGE_NODE "Back projection image" *CAMSHIFT_BACK_PROJECTION*]
Passive filter	[CAMSHIFT_RECTANGLE_NODE "Last known bounding box" *CAMSHIFT_LAST_BOX*]
Output	(x (CAMSHIFT_RECTANGLE_NODE "New bounding box") (CAMSHIFT_MOMENTS_NODE "Size and orientation"))

TABLE II

A LOGICAL LEVEL DEFINITION FOR THE CAMSHIFT CELL OF DESIGN 2 OF FIGURE 6).

Design 3 makes explicit the relationship between CAMSHIFT and the mean shift algorithm. CAMSHIFT is a feed-back loop in which the last known target location and size information (bounding box) is used to set the initial search location in the next video frame.

1) Mean shift iteratively computes the centroid of the 2D color probability distribution within the search window (*Mean shift* cell).
2) The size and orientation of the target probability distribution are computed from the zeroth and second moments of the distribution, respectively (*Moments* cell).
3) The (*Box update* cell) updates the *Last known target bounding box* information, keystone

of the feedback loop. That information becomes available to compute the initial search location for the next frame to be processed by the *Mean shift* cell.

The SAI style implies a spectrum of representations that range from all-in-code to all-in-structure encoding. The decomposition process followed here could potentially continue until each cell carries computation requiring one or few code instructions[2]. The structural patterns that occur in Design 3 capture essential properties of the system, as discussed in the next section.

B. Discussion

1) Concurrency, latency and throughput: The asynchronous semantics of SAI imply that the target information used for initialization in the mean shift cell at any given time is in fact the *latest available* known bounding box. If the latency of the feed-back loop subsystem is higher than the inverse of its throughput, the time needed to make available the position of the target in a frame is greater than the interval between two consecutive frames. The position of the target in the one before last frame will then initialize the search in the next frame. In effect, the algorithm will function under conditions similar to that of a system with lower throughput, yet will perform consistently, while maintaining the higher overall throughput. Depending on the application and the context of use, a low pass filter in the update operation will smooth out outliers and noise, and prevent divergence due to phased updates, at the cost of making the system less responsive to sudden movements.

The branching of the stream to follow separate paths, explicit encoding of causality (or lack thereof), signals that computation on the two paths are independent, and may be carried concurrently. This type of concurrency may reduce system latency. Here, the new target location should be used as soon as possible for update and for visualization, in no arbitrarily imposed order. As long as computing resources (in a general sense) are available, and assuming fair scheduling, the design will yield a system with minimal achievable latency.

2) Iterations and streams: Section II-A describes mean shift as an iterative algorithm: the same series of steps successively compute better approximate solutions, each building on the last one, until a certain level of precision is reached. An iterative algorithms is but a special case of a feedback process, in which the notion of time is abstracted, and the input is fixed. This

[2]Such a representation is not necessarily desirable in practice; expressiveness and efficiency, for example, would likely suffer.

understanding suggests a more natural and efficient integration of the optimal gradient descent that characterizes mean shift within the CAMSHIFT feedback loop.

The code structure characteristic of iterative algorithms is a loop that carries data over its iterations, and therefore cannot be trivially parallelized. The actual time required to execute the code block that includes the loop varies with the actual number of iterations, which is itself input dependent and may be highly variable. Practical implementations include a safeguard that aborts the loop if the process does not converge within a certain number of iterations. Experimentation with the CAMSHIFT demonstration system reveals that the number of iterations actually performed for each invocation of mean shift is most of the time a very small number (of the order of 2 or 3), and occasionally the maximum allowed. Intuitively, the target (a person's face in a close-up shot) is large and slow, and therefore almost always found very close to where it was previously seen. Occasionally, the target suddenly moves too fast, or disappears, or, for any other reason, cannot be found in the particular frame, and the algorithm keeps working pointlessly until the maximum number of iterations allowed is reached, wasting precious time and processing cycles on a hopeless task. Processing resources would be better spent on the next frame, in which the target might be easier to find. Consequently, the Mean shift cell of Design 3 implements a small, fixed number of iterations of the mean shift gradient descent. The iterative nature of the original algorithm is expressed through the feedback structure. As a result, the system reacts more efficiently to changes in the input (target motion).

Compared to the original black-box design (in either DirectShow or SAI), Design 3 gives an account of the operations involved in CAMSHIFT, and of their relationships and dynamics, that is more explicit to humans, and maps directly to executable code. The system design graph itself provides an informational pictorial representation of the algorithm; the picture beneficially complements the description of Section II-A. The next section illustrates the use of SAI patterns, including a generalization of the tracking pattern rendered explicit in Design 3, in composing a more complex system.

VI. Architectural Design: Stevi

This section illustrates the specialization and composition of general architectural patterns through the design of Stevi v.1, a demonstration computer vision subsystem that performs real-time people detection and tracking from stereo color video [33]. Stevi was designed and

implemented in the context of a wider project that aims to empower personal service robots with advanced vision capabilities [34]. Figure 7 shows, in the *visualization* box, a sample of Stevi's detection and tracking output (face areas and target labels) overlaid on the left and right frames of an input stereo pair. Note that the labels are consistent in left and right images.

A. System Overview

Stevi's design exemplifies a systematic approach to combining efficient but brittle algorithms into robust systems. In Stevi, monocular algorithms interact according to a general tracking pattern whose core is a feedback loop. Systems built upon such a pattern exhibit resilience properties, such as bootstrapping capability, robustness to input data noise, and adaptability to a changing environment. As a stand-alone integrated demonstration, the system incorporates video capture, processing and result visualization for evaluation purposes.

Stevi v.1's tracking core builds upon established monocular algorithms for face detection and color-based tracking to perform stereo multi-target people (head) tracking in a color stereo video stream. A classifier-based face detection algorithm based on [35] provides the core functionality for initial target acquisition. Each detected face area instantiates a target candidate to which is attached a color model (histogram) initialized from the face area in each image of the stereo pair. A mean shift-based algorithm (see Section II-A tracks the targets (or rather their color models) that have sufficient combined support in both images.

Figure 7 shows Stevi v.1's conceptual level architectural graph in SAI notation. The graph specifies the data and processing elements that implement these algorithms, as well as their relationships. Dashed boxes in the figure identify three main groups of components:

1) The *Stereo input and pre-processing* subsystem deals with the multiple camera video input and subsequent pre-processing of the video frames flowing on the stream,
2) the *Stereo multi-target detection and tracking* subsystem integrates target detection and tracking in a feedback loop pattern, and
3) the *Visualization* subsystem provides a visual insight into the tracker's state for debugging and demonstration purposes.

The following subsections outline functional and architectural specificities of the design.

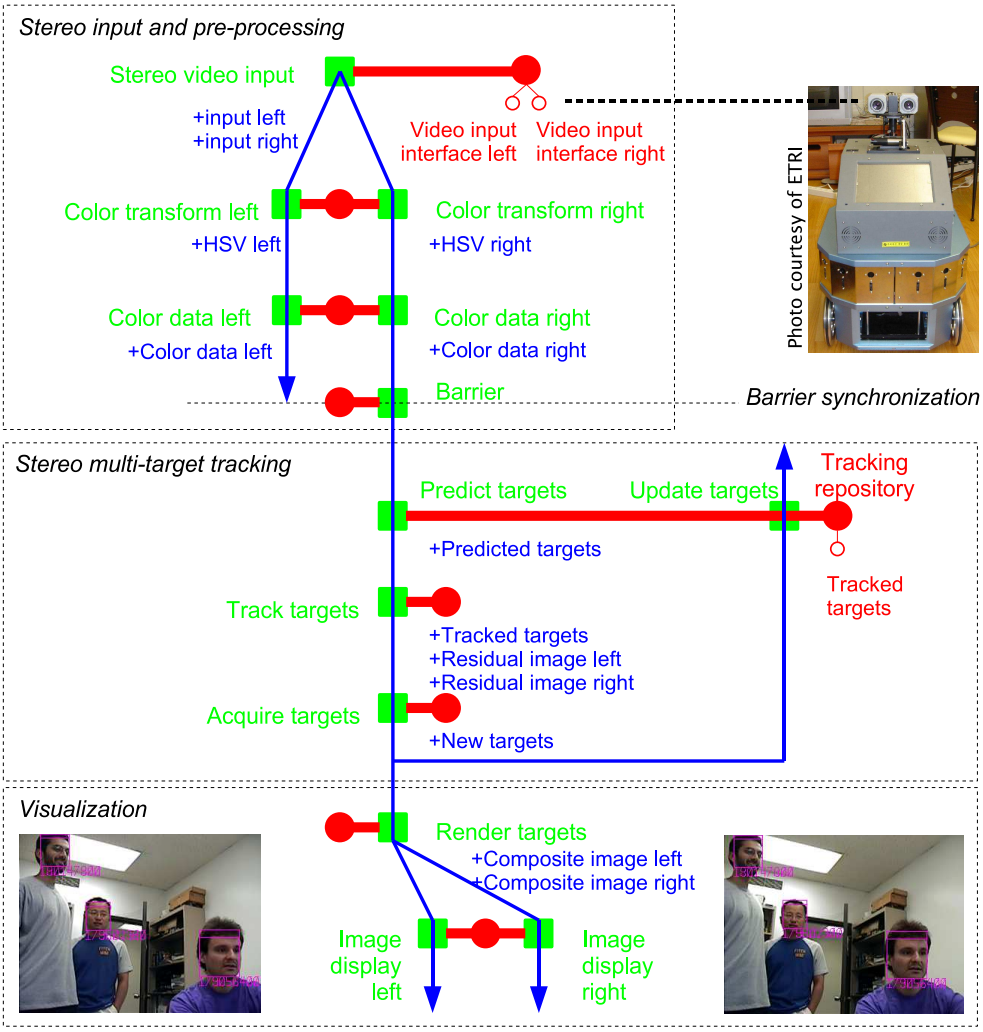

Fig. 7. Stevi v.1's conceptual level system architecture in SAI notation (adapted from [34]). The system was integrated and tested on the Weber-N prototype (picture top right). The visualization subsystem renders and displays composite images that show detection and tracking results overlaid on the left and right input images.

B. Concurrent Paths and Synchronization

The pre-processing of left and right images in each stereo pair is carried independently. The input stream, which carries pulses created by the capture of each new stereo pair of video frames, forks into two symmetric paths. Each pulse accumulates data computed from the left and right image, so that all the information related to a single pair remains synchronized. The explicit modeling of left and right processing as concurrent streams ensures that an implementation of the system will leverage the concurrency in this part of the processing, possibly resulting in

reduced latency.

Downstream processing depends on pre-processing data for both left and right paths. This dependency is expressed at the architectural level by a barrier synchronization [36] pattern, in this case a single cell and its repository. The Barrier cell holds each pulse until it contains the required data (expressed as a filter), upon which the pulse is released and transmitted normally to downstream cell(s). The MFSM middleware implements the pattern as described here. Note that this particular way of expressing the semantics of barrier synchronization is not unique, and its expression of those semantics is independent of actual implementation considerations.

C. Integrated Detection and Tracking

The subsystem delimited by the box labeled *Stereo multi-target detection and tracking* in Figure 7 instantiates a general tracking pattern that explicitly builds on the feedback nature of the task. The *Tracking repository* holds a list of target trackers. At each moment in time, the state of these trackers represents the latest known persistent information for processing. A feedback loop anchored by the *Predict targets* cell and the *Update trackers* cell accesses and maintains this information. The acquisition of each new data sample (stereo pair) triggers a cycle of the loop: The prediction process (*Predict targets* cell) generates a list of expected observations based on the current trackers. A tracking subsystem (*Track targets* cell) uses predictions and input data to produce a list of confirmed tracked target observations. A residual analysis subsystem (*Acquire targets* cell) explains input data not accounted for by the predictions, resulting in initial detection of additional targets. Finally, an updating process (*Update trackers* cell) integrates confirmed target observations into the corresponding trackers, and create new trackers for newly detected targets.

This dynamic approach to tracking is inherently robust to some issues that are difficult to handle in static or sequential approaches to online video tracking. Depending on specific design choices, a system based on the feedback pattern may exhibit properties that are out of the reach of the individual algorithms involved for detection and tracking. Because the Tracking and Acquisition subsystems operate on the same control loop, the tracking process is naturally bootstrapped: unknown targets are automatically acquired. Furthermore, the absence of observation for a given known target in a given input sample need not necessarily result in the loss of the corresponding tracker (depending on the update policy). The system is thus

resilient to a wide range of disruptions and discontinuities in the input signal as well as errors in intermediate results caused by failures of the individual algorithms.

The tracking pattern instantiated in Stevi v.1 is not specific of the particular algorithms used. It is, by design, open to the concurrent operation of multiple algorithms (e.g. for detection) and provides a formal framework for implementing fusion algorithms. The SAI formulation also opens the door to designs that compose several feedback loops to integrate and track data at different time scales and different symbolic levels. Such systems are not conveniently modeled by traditional computational approaches, and may exhibit complex dynamics. A given architecture might specify a family of systems whose behavior will depend on dynamic parameters, and whose study (and design) will then require approaches and tools that have so far been largely avoided in computer science.

D. Implementation and Performance

Stevi v.1, as specified in Figure 7, was implemented in C++ using the MFSM middleware. The running system reliably detects and tracks people at approximately 15 frames-per-seconds on a contemporary (2006) general purpose computing hardware. The software automatically takes advantage of hyper-threading and/or multiple cores when available. The system design involves established computer vision algorithms. Stevi v.1's implementation leverages face detection and color-based tracking functionalities provided by Intel's OpenCV Library. The initial Stevi prototype was developed on the MS Windows platform. The same system was ported to Linux with minimal effort, thanks to the cross-platform MFSM code and module base, and the availability of Intel's OpenCV for both platforms. Stevi v.1 was also integrated and tested on the Weber-N personal service robotic platform developed by the Electronics and Telecommunications Research Institute (prototype pictured in Figure 7).

VII. Related Work

SAI defines a new architectural framework to answer the challenges faced in the design and implementation of interactive systems. This section compares and contrasts SAI with representative related efforts.

A. Code Architecture

Code architecture is concerned with modularity, efficiency, re-use, maintainability of computer code, that is the machine representation of a system. The SAI framework aims to encode components and express essential system properties at the architectural level, where they can be represented and manipulated at a higher level of abstraction than at the programming language level (as advocated for example by Parnas in [37]).

Code libraries developed within an application domain, often assume an underlying (system) architectural model that is convenient in the specific application context. The implicit adoption of a domain-specific architectural style hinders the applicability of libraries across application domain boundaries.

B. Software Architectures

Shaw and Garlan [27] define software architecture as a level of design that "involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition and constraints on these patterns." The SAI architectural style clearly fits within this framework, and can be interpreted as a hybrid style combining features of classical styles such as dataflow (e.g. Pipes-And-Filters), data-centered (e.g. Blackboard), event-based communicating components, etc.

Software architecture research has produced domain specific styles, such as the C2 style for Graphical User Interfaces [38]. Software architecture concerns from within application fields have led to the adoption and specialization of classical styles. Architectures for multimedia streaming systems include MIT's VuSystem [39], the Berkeley Continuous Media Toolkit [40], the Network Integrated Media Middleware [41], and the Distributed Media Journaling (DMJ) project [42] [43]. These efforts adopt modular dataflow architecture concepts. They are designed primarily for audio and video on-line processing and transmission, with a strong emphasis on capture, transmission and replay aspects. The datastream processing model, however, is not well suited to the design of interactive media content creation tools (and other interactive software), for which windowing message-based frameworks have been the norm. The model underlying interaction tools is usually data-driven. For example, the "pool of frames" approach to temporal media data handling in the MVC architecture [44] represents a video stream as as an object whose components (frames) can be edited in a random access fashion, not as a sequential stream of data.

In general, symbolic computations (as encountered for example in Artificial Intelligence) involve the manipulation of dynamic but persistent data structures, while signal processing techniques are concerned with data streams.

In the field of music computing, the co-dependency between composition and performance has exposed the fundamental divide between processing and representation paradigms [45]. The difficulty of combining (functional) signal processing and (imperative) event processing has prompted the development of original code architectures combining interaction and real-time signal synthesis [46].

Robotics' three-layer architectures [47] [48] reflect similar concerns: the "low-level" layer is concerned with real-time sensing and actuation (signal processing), the "high-level" layer handles symbolic (and non real-time) processing such as planning, and an intermediate layer ensures the low and high level layers cohabitate and communicate smoothly in an integrated system. In practice, the architectural styles (or models of computation) that underly the low and high-level layers are well understood, but incompatible, and the challenge, as yet unsolved in the general case, resides in the architectural design of the middle layer, and of its interactions with the two other layers. SAI's unified model allows to use the same formalism to design not only low- and high-level layer elements, but also, and perhaps more importantly, everything in between.

Recently, video games-themed research has permeated a number of academic computer science and engineering fields. This trend has generated little academic work regarding methodologies, models and tools for designing the complex interactive systems of which video games are popular instantiations. Software efforts focus on applying "good practices" of code architecture in the design of code libraries that implement mostly graphics features re-targeted and augmented for interaction [49] [50]. Software system architecture is hardly mentioned; "3-D engines" anchored by an infinite rendering / event processing loop remain the norm. Scalability and concurrency issues remain difficult challenges, especially in the context of multi-core hardware architectures. Interactive systems designed in the SAI style (such as, for example, the interactive music systems listed in section I-B replace the one event processing loop with concurrent streams that interact (and synchronize when necessary) with each other through persistent data structures. This approach yields more efficient, interactive, and naturally concurrent systems.

C. Architecture Description Languages

The SAI framework offers features that go beyond the definition of an architectural style, to include some features of Architecture Description Languages (ADLs). The purpose of ADLs is to allow architectural descriptions that aid the understanding of, and communication about, software systems. For this purpose, the minimal features of an ADL include a simple syntax (possibly graphical), well understood semantics, simple tools for visualization, understanding, and simple analysis of architectural descriptions. Many ADLs offer advanced features such as formal semantics and powerful analysis tools. ADLs vary in their domains of application, levels of abstraction and specificity, syntax, etc. Medvidovic and Taylor describe a classification and comparison framework for ADLs, and a thorough analysis, in [51]. Representative example ADLs include Rapide [52] [53], C2 [54], SADL [55] [56], and Wright [57] [58]. Unifying afforts include ACME [59], which originated as an architectural interchange language, and Mehta and Medvidovic's (Alfa) [60], which defines a set of primitives for composing architectural styles. Preliminary explorations have successfully modeled traditional style elements (i.e. component, connectors and constraints) as SAI architectural patterns. Figure 8 offers four simple examples (from [27]). The SAI formulation reveals that feed-back (c) and feed-forward (d) follow very similar architectural patterns, and makes explicit the process dependency that differentiates the two patterns. The expression of different styles with SAI primitives allows the instantiation and integration of the corresponding patterns in designs that also model the interactions between them. The framework also provides a direct path from design to implementation of such systems.

Integrated visual design and analysis environments will fully leverage SAI's graphical notation, as well as algorithms for automatic checking of conformance to style, liveness and safety (at the architectural level), which are under development. The VisualSAI [61] project has produced a prototype of such an environment, implemented as plug-in for the Eclipse platform [62]. Other functionalities under development include automatic code generation, targeted at the MFSM middleware.

D. Models for Concurrent Computing:

Concurrent use of shared resources (e.g. memory) is the source of major difficulties in code design and programming, and a major feature of the SAI model. Race conditions cause unpredictability of system behavior. Mutual exclusion prevents race conditions, but can introduce

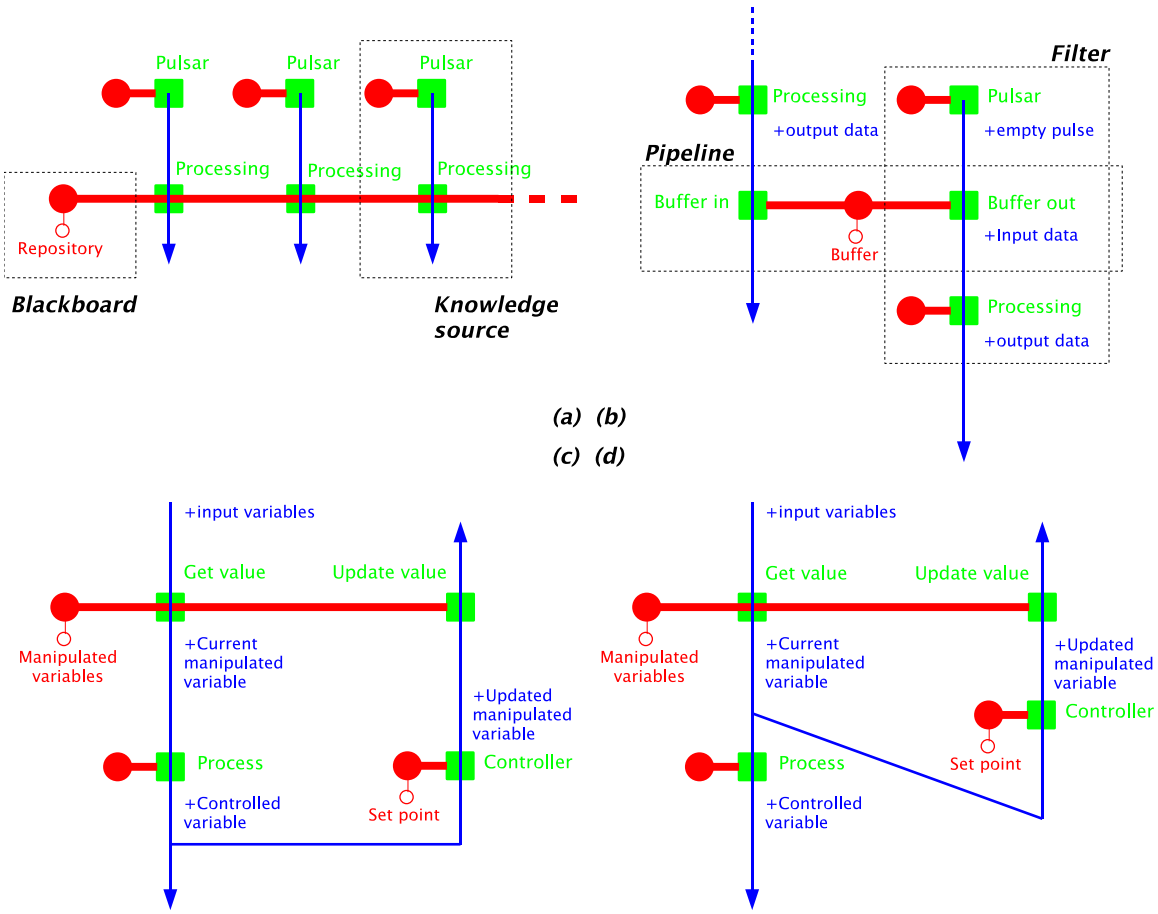

Fig. 8. Architectural styles as SAI patterns. (a) Parallel asynchronous processing underlies *Blackboard* and other shared repository styles. (b) Sequentiality is at the core of the *Pipes and Filters* style: data samples must be buffered in a First-In-First-Out queue in order to enforce sequentiality of processing. (c) Feed-back: the controller modifies the values of the manipulated variables based on the value of the controlled variable output by the process. (d) Feed-forward: the controller modifies the values of the manipulated variables based on the values of some input variables to the process.

other problems such as deadlock and starvation. Most formalisms therefore enforce a message-passing communication scheme.

Process calculi, whose representatives include CSP [63] [64], CCS [65], ACP [66], and Pi-calculus [67], define primitives and operators to combine them in order to describe processes and systems. They also define algebraic laws for the process operators, which allow process expressions to be manipulated using equational reasoning. These formalisms make abstraction of time (which makes modeling dynamic systems difficult), and impose synchronous message

passing. In contrast, the SAI model makes qualitative and quantitative time explicitly an inherent part of the processing and data model, and the message passing communication is deliberately asynchronous.

The actor model [68] [69] [70] is another message passing model for concurrent computation. Actors exchange messages asynchronously (only mode of communication). Computation within and among actors is inherently concurrent. SAI adopts a similarly asynchronous (unbuffered) message passing communication model for volatile data. The model does not impose any requirements on the order of arrival of messages, but time stamps and structures allow buffering/reordering if (and only if) needed.

The Ptolemy Project [71] is concerned with interaction of concurrent components involving heterogeneous mixtures of models of computations, in the context of (hard) real-time systems. Ptolemy focuses on timing and synchronization issues. Rather, SAI provides a general, unifying formalism that allows to explicitate the similarities between, and singularities of, specific existing models that address subsets of those requirements, independently and at different levels of abstraction.

VIII. Summary and Future Work

This article presented the SAI framework for the design, analysis and implementation of interactive software systems. SAI defines a formal architectural style that combines an extensible data model and a hybrid (shared memory and message-passing) distributed asynchronous concurrent processing model, to allow natural and efficient manipulation of data streams. The modularity and scalability of the style facilitate distributed code development, testing, and reuse, as well as fast system design and integration, maintenance and evolution. SAI promotes the encoding of system logic in the structural organization of simple computing components, rather than in the complexity of the computations carried out by individual components. SAI designs exhibit a rich variety of structural and functional *architectural patterns*, suitable for systematic study and re-use. A graph-based notation for architectural designs affords intuitive and scalable system representation at the conceptual and logical levels. SAI allows a continuum of intermediate-level representations from conceptual to logical to physical specifications, through the use of an appropriate architectural middleware, such as the open source Modular Flow Scheduling Middleware (MFSM). Systems designed with SAI and implemented with MFSM are automatically multi-threaded, and can

directly take advantage of multiple CPUs, hyper-threading and multi-core processor architectures.

Example system from the field of computer vision helped illustrate and ground the concepts and principles that drive the definition and use of the SAI framework. SAI has been an engine for innovation in cross-disciplinary projects and in the classroom. The SAI framework has been used in the design and implementation of numerous interactive systems in various fields including computer vision, graphics, and music; the framework was instrumental in the successful implementation of experimental courses in software development, graduate and undergraduate, that pool the efforts of the entire class on a single, ambitious collaborative project.

Current research focuses on further formalization of the framework and its underlying principles. For example, a systematic study could lead to the construction of an ontology of structural patterns. The characterization of functional patterns within this ontology could prove a useful resource for guiding system design. Automatic code generation, automatic methods for proving properties of system designs expressed in SAI (such as correctness, safety and liveness), as well as high-level pattern-based design analysis tools, will progressively enrich SAI-based visual design environments.

ACKNOWLEDGEMENTS

This article was completed at the Radcliffe Institute for Advanced Study at Harvard University. Most of the work described was conducted at the University of Southern California with generous support from various internal and external funding sources. The SAI project was supported in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152. The CAMSHIFT tracker study was supported in part by the Advanced Research and Development Activity of the U.S. Government under contract No. MDA-904-03-C-1786. The STEVI system was created in the context of a collaborative project with the Electronics and Telecommunications Research Institute, a Korean non-profit, government-funded research organization. Any Opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the above mentioned sponsors.

REFERENCES

[1] D. McLeod, U. Neumann, C. Nikias, and A. Sawchuk, “Integrated media systems.” *IEEE Signal Processing Mag.*, vol. 16, no. 1, pp. 33–43, Jan. 1999.

[2] A. R. François, "Software architecture for immersipresence," Integrated Media Systems Center, University of Southern California, Tech. Rep. IMSC-03-001, December 2003.

[3] ——, "A hybrid architectural style for distributed parallel processing of generic data streams," in *Proceedings of the International Conference on Software Engineering*, Edinburgh, Scotland, UK, May 2004, pp. 367–376.

[4] ——, "Modular Flow Scheduling Middleware (MFSM) project." [Online]. Available: mfsm.sourceforge.net

[5] ——, "Software Architecture for Computer Vision," in *Emerging Topics in Computer Vision*, G. Medioni and S. Kang, Eds. Upper Saddle River, NJ: Prentice Hall, 2005, pp. 585–654.

[6] A. R. François and E. Kang, "A handheld mirror simulation," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, vol. II, Baltimore, MD, July 2003, pp. pp. 745–748.

[7] A. François, E. Kang, and U. Malesci, "A handheld virtual mirror," in *ACM SIGGRAPH Conference Abstracts and Applications proceedings*, San Antonio, TX, July 2002, p. 140.

[8] A. R. François, "Components for immersion," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, Lausanne, Switzerland, August 2002.

[9] A. R. François and E. Chew, "An architectural framework for interactive music systems," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Paris, France, June 2006, pp. 150–155.

[10] E. Chew and A. R. François, "Interactive multi-scale visualizations of tonal evolution in MuSA.RT opus 2," *ACM Computers in Entertainment*, vol. 2, no. 4, October-December 2005.

[11] ——, "MuSA.RT - Music on the Spiral Array . Real-Time," in *Proceedings of ACM Multimedia*. Berkeley, CA, USA: Sheridan Printing Co. Inc., November 2-8 2003.

[12] E. Chew, A. R. François, J. Liu, and A. Yang, "ESP: A Driving Interface for Expression Synthesis," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Vancouver, B.C., Canada, May 2005.

[13] J. Liu, E. Chew, and A. R. François, "From driving to expressive music performance: Ensuring tempo smoothness," in *Proceedings of the ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, Hollywood, CA, USA, June 2006.

[14] E. Chew, J. Liu, and A. R. François, "ESP: Roadmaps as constructed interpretations and guides to expressive performance," in *Proceedings of the ACM Workshop on Music and Audio Computing*, Santa Barbara, CA, USA, October 2006.

[15] A. R. François, E. Chew, and D. Thurmond, "Visual feedback in performer-machine interaction for musical improvisation," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, New York, NY, USA, June 2007, pp. 277–280.

[16] A. R. François, "CAMSHIFT tracker design experiments with Intel OpenCV and SAI," Institute for Robotics and Intelligent Systems, University of Southern California, Tech. Rep. IRIS-04-423, July 2004.

[17] A. R. François and R. Zimmermann, "csci599 - Integrated Media Systems (Fall 2002)." [Online]. Available: www-scf.usc.edu/~csci599z

[18] A. R. François, "csci201g - Principles of Software development - Games." [Online]. Available: www-scf.usc.edu/~csci201g

[19] "Crosswinds." [Online]. Available: www-scf.usc.edu/~csci201g/Sp2007/project.html

[20] G. R. Bradski, "Computer video face tracking for use in a perceptual user interface," *Intel Technology Journal*, Q2 1998.

[21] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, May 2002.

[22] D. Comaniciu, V. Ramesh, and P. Meer, "Real-time tracking of non-rigid objects using mean shift," in *Proceedings of the Int. Conf. Computer Vision and Pattern Recognition*, 2000, pp. II: 142–149.

[23] "Intel Open Source Computer Vision Library." [Online]. Available: www.intel.com/research/mrl/research/opencv

[24] G. R. Bradski, "The OpenCV Library," *Dr. Dobb's Software Tools for the Professional Programmer*, November 2000.

[25] M. Turk and M. Kölsch, "Perceptual Interfaces," in *Emerging Topics in Computer Vision*, G. Medioni and S. Kang, Eds. Upper Saddle River, NJ: Prentice Hall, 2005, pp. 455–519.

[26] "Microsot DirectShow documentation." [Online]. Available: msdn.microsoft.com/en-us/library/ms783323.aspx

[27] M. Shaw and D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996.

[28] A. R. François, "Web-Cam Computer Vision (WCCV) project." [Online]. Available: wccv.sourceforge.net

[29] ——, "Open Virtuality Engine (OpenVE) project." [Online]. Available: openve.sourceforge.net

[30] "OpenGL." [Online]. Available: www.opengl.org

[31] "OpenGL Utility Toolkit (GLUT)." [Online]. Available: www.opengl.org/resources/libraries/glut

[32] A. R. François and G. G. Medioni, "A modular software architecture for real-time video processing," in *IEEE International Workshop on Computer Vision Systems*. Vancouver, B.C., Canada: Springer-Verlag, July 2001, pp. 35–49.

[33] ——, "A vision system for personal service robots: Resilient detection and tracking of people," Computer Science Department, University of Southern California, Tech. Rep. 06-880, June 2006.

[34] G. G. Medioni, A. R. François, M. Siddiqui, K. Kim, and H. Yoon, "Robust real-time vision for a personal service robot," *Computer Vision and Image Understanding*, vol. 108, no. 1-2, pp. 196–203, 2007.

[35] P. Viola and M. J. Jones, "Robust real-time face detection," *International Journal of Computer Vision*, vol. 57, no. 2, pp. 137–154, 2004.

[36] G. R. Andrews, *Foundations of multithreaded, parallel and distributed programming*. Addison Wesley, 2000.

[37] P. Devanbu, B. Balzer, D. Batory, G. Kiczales, J. Launchbury, D. Parnas, and P. Tarr, "Modularity in the new millenium: A panel summary," in *Proceedings of the International Conference on Software Engineering*, Portland, OR, USA, May 2003, pp. 723–724.

[38] R. N. Taylor, N. Medvidovic, K. M. Anderson, J. E. James Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow, "A component- and message-based architectural style for gui software," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, June 1996.

[39] C. Lindblad and D. Tennenhouse, "The VuSystem: A programming system for compute-intensive multimedia," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1298–1313, Sept. 1996.

[40] K. Mayer-Patel and L. Rowe, "Design and performance of the Berkeley Continuous Media Toolkit," in *Multimedia Computing and Networking*, M. Freeman, P. Jardetzky, and H. Vin, Eds., 1997, pp. 194–206.

[41] M. Lohse, M. Repplinger, and P. Slusallek, "An open middleware architecture for network-integrated multimedia," in *Proceedings of the Joint International Workshop on Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems (IDMS/PROMS 2002)*, Coimbra, Portugal, 2002.

[42] V. Eide, F. Eliassen, O.-C. Granmo, and O. Lysne, "Scalable independent multi-level distribution in multimedia content analysis," in *Proceedings of the Joint International Workshop on Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems (IDMS/PROMS 2002)*, Coimbra, Portugal, 2002.

[43] V. S. W. Eide, F. Eliassen, O.-C. Granmo, and O. Lysne, "Supporting timeliness and accuracy in distributed real-time content-based video analysis," in *Proceedings of the ACM International Conference on Multimedia*, Berkeley, California, USA, November 2003, pp. 21–32.

[44] N. T. Graham and T. Urnes, "Integrating support for temporal media into an architecture for graphical user interfaces," in *Proceedings of the International Conference on Software Engineering*, Boston, MA, 1997.

[45] M. S. Puckette, "A divide between 'compositional' and 'performative' aspects of Pd," in *Proc. First International Pd Convention*, Graz, Austria, 2004.

[46] R. B. Dannenberg, "A language for interactive audio applications," in *Proc. International Computer Music Conference*, San Francisco, CA, 2002.

[47] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, pp. 14–23, April 1986.

[48] E. Gat, "On three-layer architectures," in *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. Bonnasso, and R. Murphy, Eds. AAAI Press, 1998.

[49] D. H. Eberly, *3D Game Engine Design : A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann, 2000.

[50] ——, *3D Game Engine Architecture : Engineering Real-Time Applications with Wild Magic*. Morgan Kaufmann, 2005.

[51] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, January 2000.

[52] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336–355, April 1995.

[53] D. C. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, September 1995.

[54] D. S. R. Nenad Medvidovic and R. N. Taylor, "A language and environment for architecture-based software development," in *Proceedings of the International Conference on Software Engineering*, Los Angeles, CA, USA, May 1999, pp. 44–53.

[55] M. Moriconi and R. A. Riemenschneider, "Introduction to sadl 1.0: A language for specifying software architecture hierarchies," SRI International, Tech. Rep. SRI-CSL-97-01, March 1997.

[56] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct architecture refinement," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 356–372, April 1995.

[57] R. Allen, "A formal approach to software architecture," Ph.D. Thesis, Carnegie Mellon University, May 1997.

[58] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, july 1997.

[59] D. Garlan, R. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of CASCON'97*, Toronto, Ontario, Canada, November 1997, pp. 169–183.

[60] N. R. Mehta and N. Medvidovic, "Composing architectural styles from architectural primitives," in *Proceedings of the European Software Engineering Conference - ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Helsinki, Finland, September 2003, pp. 347–350.

[61] A. R. François, "VisualSAI project." [Online]. Available: visualsai.sourceforge.net

[62] "Eclipse." [Online]. Available: www.eclipse.org/

[63] C. A. Hoare, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, 1985.

[64] L. Lamport and F. B. Schneider, "The "hoare logic" of csp, and all that," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, pp. 281–296, 1984.

[65] R. Milner, *A calculus of communicating systems*. Springer Verlag, 1980.

[66] J. A. Bergstra and J. W. Klop, "Act: A universal axiom system for process specification," *Algebraic Methods*, pp. 447–463, 1987.

[67] R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*. Springer Verlag, 1999.

[68] C. E. Hewitt, "Viewing control structures as patterns of passing messages," *Journal of Artificial Intelligence*, vol. 8, no. 3, pp. 323–363, 1977.

[69] G. A. Agha, *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[70] ——, "Abstracting interaction patterns: a programming paradigm for open distributed systems," in *Formal methods for open object-based distributed systems*, E. Najm and J.-B. stefani, Eds. Chapman & Hall, 1997.

[71] E. A. Lee, "Ptolemy project." [Online]. Available: ptolemy.eecs.berkeley.edu