CAMSHIFT Tracker Design Experiments with Intel OpenCV and SAI

Alexandre R.J. François
Institute for Robotics and Intelligent Systems
University of Southern California
afrancoi@usc.edu

July 2004

Abstract

When humans interact with computer systems, they expect the experience to meet human standards of reactiveness, robustness and, if possible, non-intrusiveness. In order for computer vision techniques to have a significant impact in human-computer interaction, the development of efficient and robust algorithms, as well as their integration and operation as part of complex (including multi-modal) systems, must be specifically addressed. This report describes design and implementation experiments for CAMSHIFT-based tracking systems using Intel's Open Computer Vision library and SAI (Software Architecture for Immersipresence), a software architecture model created specifically to address the integration of different solutions to technical challenges, developed independently in separate fields, into working systems, that operate under hard performance constraints. Results show that the SAI formalism is an enabling tool for designing, describing and implementing robust systems of efficient algorithms.

Keywords: Software Architecture, Perceptual User Interface, Human-Computer Interaction.

1 Introduction

When humans interact with computer systems, they expect the experience to meet human standards of reactiveness, robustness and, if possible, non-intrusiveness. Reactiveness can be expressed in terms of perceived system latency (delay between a user's action and the perception of the action's effect on the system). Perceived latency results from the actual latencies and throughputs of the various processes involved in the system and their relationships. Robustness refers to the system's ability to cope with unexpected situations.

Non-intrusive Human-Computer Interaction (HCI) modalities are regrouped under the term Perceptual User Interfaces (PUIs) [13], a field in which computer vision should find ample application. Many image and video processing algorithms are now available that can be implemented to operate in real-time. However, simplicity and robustness seem mutually exclusive, and vision systems that fulfill both reactiveness and robustness requirements are very few. Efforts to improve robustness of simple and efficient techniques usually result in more complex and over-specialized algorithms that are not well suited for use in real-time systems (even with the help of Moore's law). The work reported here is driven, in part, by the belief that computer vision performance on par with human expectations and abilities will be achieved by designing and implementing *robust systems of efficient (but fallible) algorithms.*

In mainstream Computer Vision, the jump from algorithm to system is often taken for granted and over-simplified. Most published algorithms are tested in proof-of-concept systems whose design is not given much consideration. Intel's Open Computer Vision library [2] regroups a large collection of standard data structures and efficient implementations of computer vision algorithms. How these algorithms may be used to design and implement real applications, or software systems, is out of the scope of the library. Among the various models available to programmers, dataflow architectures, an example of which is Microsoft's DirectShow architecture [11], have become popular for video processing systems. Dataflow models however are not suitable for all types of applications, and in fact are particularly ill-suited for the design of interactive systems [12]. In order for computer vision techniques to have a significant impact in HCI in general, and PUIs in particular, their integration and operation as part of complex (including multi-modal) systems must be specifically addressed.

SAI (Software Architecture for Immersipresence) [8] is a software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams. SAI was created specifically to address the integration of different solutions to technical challenges, developed independently in separate fields, into working systems, that operate under hard performance constraints.

This paper reports design and implementation experiments for CAMSHIFT-based tracking systems using OpenCV and SAI. The goal of these experiments is not to evaluate the suitability of a particular algorithm for a particular purpose, but rather to investigate the use of a recently introduced component architecture model and available implementation tools for designing and understanding complex systems. Section 2 provides short overviews of CAMSHIFT, dataflow architectures and SAI. Section 3 presents experiments in designing a CAMSHIFT-based tracking system with SAI. Section 4 addresses implementation of the proposed designs and presents quantitative development and performance comparative analysis. Section 5 offers a summary of the contributions, a discussion and some perspectives on future research directions suggested by the results.

2 Background and Context

2.1 CAMSHIFT in OpenCV

The Continuously Adaptive Mean SHIFT (CAMSHIFT) algorithm [4], is based on the mean shift algorithm [5], a robust non-parametric iterative technique for finding the mode of probability distributions.

Given a color image and a color histogram, the image produced from the original color image by using the histogram as a look-up table is called back-projection image. If the histogram is a model density distribution, then the back projection image is a probability distribution of the model in the color image. CAMSHIFT detects the mode in the probability distribution image by applying mean shift while dynamically adjusting the parameters of the target distribution. In a single image, the process is iterated until convergence (or until an upper bound on the number of iterations is reached).

A detection algorithm can be applied to successive frames of a video sequence to track a single target. The search area can be restricted around the last known position of the target, resulting in possibly large computational savings. This type of scheme introduces a feed-back loop, in which the result of the detection is used as input to the next detection process. The version of CAMSHIFT applying these concepts to tracking of a single target in a video stream is called Coupled CAMSHIFT.

The Coupled CAMSHIFT algorithm as described in [4] is demonstrated in a real-time head tracking application, which is part of the Intel OpenCV library. The area of application specifically considered is that of PUIs, 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. Best results are obtained for skin area tracking when using the Hue component in the color images and the histogram. In the remainder of this paper, CAMSHIFT refers to the coupled CAMSHIFT algorithm, as implemented in OpenCV.

OpenCV contains features useful in implementing a CAMSHIFT-based tracking system, that will be explored in the experiments described in this paper. These features range from general purpose data structures (e.g. histograms) and functions (e.g. color conversion, back projection), to more specialized functions (e.g. MeanShift, CamShift) that package calls to general purpose functions in specific algorithms, to a black-box CAMSHIFT tracker class that encapsulates all necessary data structures and hides all intermediate library calls internally. Although it can make programming easier, such a neatly packaged programming component still requires something else to build a working system: an architecture model.

2.2 Architectural considerations

Software Architecture is the field of study concerned with the design, analysis and implementation of software systems [12]. A specific architecture can be described as a set of computational *components*, and their interrelations, or *connectors*. An *architectural style* characterizes families of architectures that share some patterns of structural organization.

Whether explicitly or implicitly, any implementation of a computer program follows some architectural model. The CAMSHIFT demonstration that comes with OpenCV is designed and implemented using Microsoft's DirectShow dataflow architecture and library. Principles common to dataflow architectures form the Pipes and Filters architectural style. It is particularly popular in the domains of signal processing, parallel processing and distributed processing [3], to name but a few.

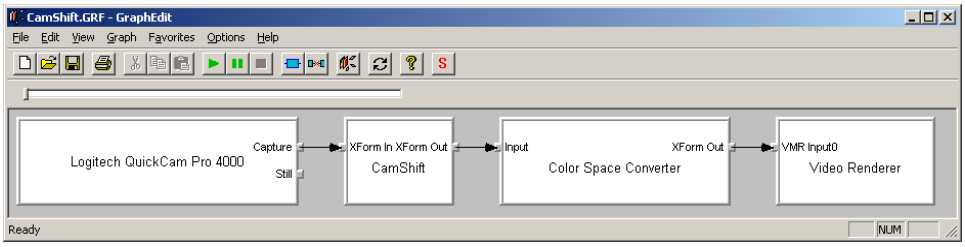

Figure 1: DirectShow graph for the OpenCV CAMSHIFT tracker demonstration.

In the Pipes and Filters style, the components, called *filters* have a set of inputs and a set of outputs. Each component reads (and consumes) a stream of data on its input and produces a stream of data on its output. The connectors are *pipes* in which the output stream of a filter flows to the input of the next filter. Filters can be characterized by their input/output behavior: source filters produce a stream without any input; transform filters consume an input stream and produce an output stream; sink filters consume an input stream but do not produce any output. The OpenCV CAMSHIFT demonstration comprises three main filters (see figure 1: a video source, a CamShift transform filter that encapsulates the OpenCV CAMSHIFT tracker class, and a video renderer sink. An additional color space conversion filter is required for the renderer to accept the output of the CamShift filter as input.

The Pipes and Filters style has a number of desirable properties that make it an attractive and efficient model for a number of applications. It is relatively simple to describe, understand and implement. The localization and isolation of computations facilitates system design, implementation, maintenance and evolution. Reversely, filters can be implemented and tested separately. Furthermore, because filters are independent, the model naturally supports parallel and distributed processing. Dataflow models are also quite intuitive, and allow to model systems while preserving the flow of data. Because of the well-defined and constrained interaction modalities between filters, complex systems are easily understandable as a series of well defined local transformations. This property is obviously not fully leveraged in the CAMSHIFT demonstration, as the tracking algorithm is implemented as a single filter.

In spite of these positive aspects, dataflow models have well studied limitations that make them unsuitable for designing multi-modal, interactive systems. Generally, most systems are designed in different styles, introducing unmapped areas at the interface between the resulting parts. A unifying hybrid architecture model, SAI, was introduced to allow consistent design and modeling of complex systems.

2.3 Overview of SAI

SAI is a software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams. SAI provides a universal framework for the distributed implementation of algorithms and their easy integration into complex systems that exhibit desirable software engineering qualities such as efficiency, scalability, extensibility, reusability and interoperability.

SAI defines an architectural style, whose underlying extensible data model and hybrid (shared memory and message-passing) distributed asynchronous parallel processing model allow natural and efficient manipulation of generic data streams, using existing libraries or native code alike. SAI provides formal, system design-oriented architectural support compatible with agile methodology practices. The modularity of the style facilitates distributed code development, testing, and reuse, as well as fast system design and integration, maintenance and evolution. The underlying asynchronous parallel processing model allows to specify optimal designs in terms of throughput and latency.

A graph-based notation for architectural designs allows intuitive system representation at the conceptual and logical levels, while at the same time mapping closely to the physical level. Figure 2 presents an overview of SAI defining elements in their standard notation.

In SAI, all data is encapsulated in pulses. A pulse is a carrier for all the synchronous data corresponding to a given time stamp. Information in a pulse is organized as a mono-rooted composition hierarchy of *node* instances. The nodes constitute an extensible set of atomic data units that implement or encapsulate specific data structures. Pulses holding volatile data flow down streams defined by connections between processing centers called *cells*, in a message passing fashion. They trigger computations, and are thus called *active*

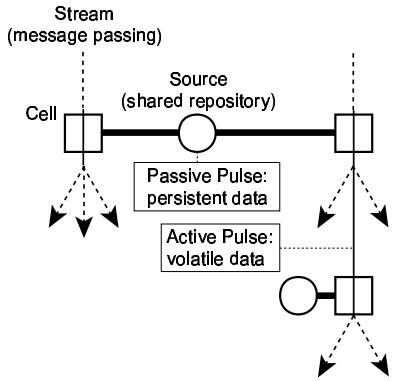

Figure 2: Overview of SAI elements. Cells are represented as squares, sources as circles. Source-cell connections are drawn as fat lines, while cell-cell connections are drawn as thin arrows crossing over the cells.

pulses. In contrast, pulses holding persistent information are held in repositories called *sources*, where the processing centers can access them in a concurrent shared memory access fashion. Processing in a cell may result in the augmentation of the active pulse (input data), and/or update of the passive pulse (process parameters). The processing of active pulses is carried in parallel, as they are received by the cell. This hybrid model combining message passing and shared repository communication, combined with a unified data model, provides a universal processing framework.

A particular system architecture is specified at the conceptual level by a set of source and cell instances, and their inter-connections. The specialized cells may be accompanied by a description of the task they implement. 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.

3 System design experiments

This section reports experiments in designing a CAMSHIFT-based tracking system with SAI. A reverse engineering approach, starting from the OpenCV CAMSHIFT demonstration, and getting deeper into the details of CAMSHIFT, produces designs of increasing apparent complexity but also increasing explicitness.

3.1 The CAMSHIFT demonstration in SAI

The most immediate way to implement a CAMSHIFT-based tracking system with OpenCV is to use the `CvCamShiftTracker` class as a black-box, mirroring its use in the DirectShow filter. Figure 3 shows the resulting design (*design 1*). Just like the CAMSHIFT specific portion of the OpenCV CAMSHIFT demonstration is completely encapsulated in a single DirectShow filter, it is encoded as a minimal functional unit in SAI. This unit, composed of a single source and a single cell, is delineated by a dashed line in the figure. Standard video input and image display modules provide capture and display functionalities.

One defining feature of the SAI framework is the explicit identification of two fundamental data classes: *volatile* and *persistent*. Volatile data is used, produced and/or consumed, and remains in a system only for a limited fraction of its lifetime. For example, in a video processing application, the video frames captured and processed are typically volatile data: after they have been processed and displayed or saved, they are not kept in the system. Process parameters, on the other hand, must remain in the system for the whole duration of its activity.

In design 1, an instance of the tracker class encodes both processing functions and persistent data, and should thus be held in a source. A custom node type encapsulates an instance of the `CvCamShiftTracker` class. The input and output frames are volatile. The custom CAMSHIFT tracker cell feeds each input frame to the persistent instance of `CvCamShiftTracker`, and creates an output synthetic frame with the result of its processing. The synthetic frame is added to the pulse and subsequently displayed. Because the same persistent instance of `CvCamShiftTracker` is accessed concurrently by individual threads processing subsequent input frames, the node encapsulating the object must also provide a mutual exclusion mechanism

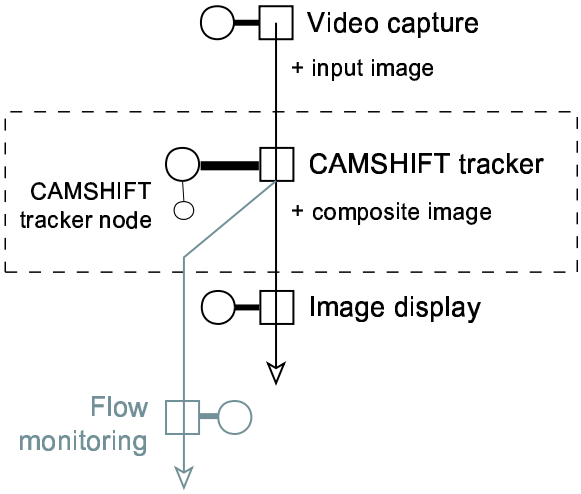

Figure 3: Design 1: black-box approach. The dashed line delineates the CAMSHIFT functional unit. A flow monitoring unit allows to collect throughput and latency statistics with minimum impact on overall system performance.

to ensure computational consistency. As a result, the processing of each input frame is performed within a critical section, resulting in a sequentialization of this part of the system.

3.2 Explicit tracker architecture

Design 1, although correct and straightforward, is not really satisfactory. First, the output of the tracker cell is an image, which is useful for visualizing the results of the tracker, but definitely not the form in which they would be used in a PUI-based system. This aspect is easily solved by limiting the CAMSHIFT tracker cell to only tracking, and having a separate custom cell perform the rendering of the synthetic image.

Even then, having a single persistent object carry all the computations in a critical section without a second look is, at least potentially, not the most efficient use of SAI's asynchronous parallel processing model. The sequentialization of the processing in this part of the system re-correlates throughput and latency, a potentially serious performance bottleneck. The modularity of SAI could certainly be better used.

An analysis of the code for the `CvCamShiftTracker` class shows a three-step processing sequence for each input frame.

First, the input RGB color image is converted to a equivalent representation in the HSV color space. This step is independent of subsequent processing, and can be performed in parallel on successive input frames. OpenCV provides the `cvCvtColor` function for color space conversion.

Second, the HSV data is used in conjunction with a color histogram in a process called back-projection, which essentially produces a color probability image from the input color image. The back projection image encodes for each pixel, its probability of belonging to the color probability distribution represented by the histogram. The histogram itself must be initialized by sampling a representative area of one frame, specified manually. The back-projection operation is independent of subsequent processing. The only persistent data used is the color histogram. OpenCV provides data structures for histograms and associated functions, including `cvCalcBackProject`.

Finally, the core of the CAMSHIFT algorithm itself is applied, coded as a call to the `cvCamShift` function. Given an initial search window and a back-projection image, the function returns the bounding box of the most probable detection area in the image, and a size and orientation estimate of the distribution. The bounding box is used to infer the initial search window in the next input back-projection image. The back projection image is obviously volatile data. The initial search window data (input) is related to the detected area (output) in the previous frame. In an asynchronous approach, this information can be encoded as the last known detection result (bounding box), which is persistent information updated after the processing of each new frame, thus forming a feed-back loop.

Figure 4 shows an SAI design (*design 2*) making these three independent steps (color conversion, back-projection and CAMSHIFT) explicit, with a separate rendering unit to produce the result visualization

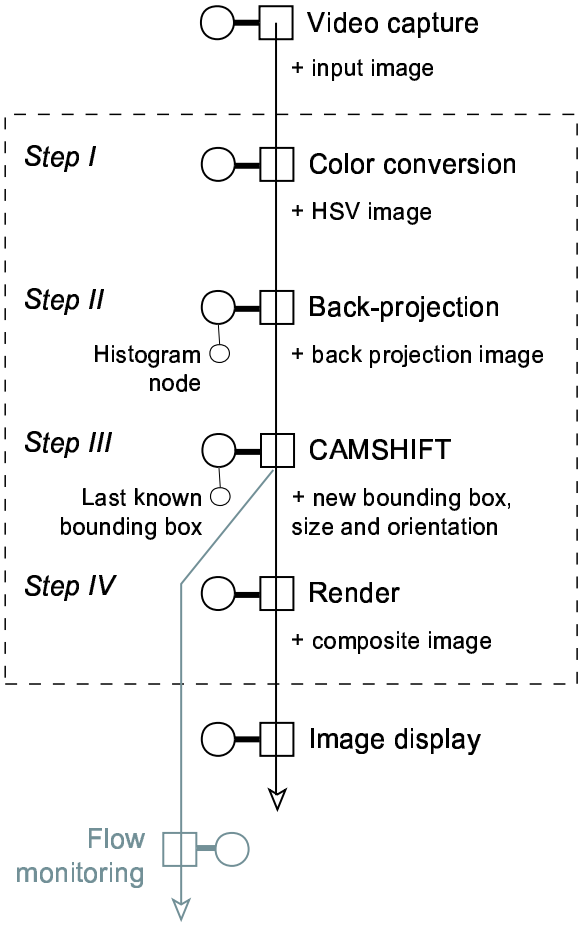

Figure 4: A 3(+1)-step design. The dashed line delineates the subsystem functionally equivalent to that delineated in figure 3. A flow monitoring unit allows to collect throughput and latency statistics with minimum impact on overall system performance.

images.

The composition of the four units in a straightforward process dependency sequence is functionally equivalent to the single CAMSHIFT source/cell unit of design 1. Because independent processes are separated, this design is more efficient than the single-unit design. Instead of a single critical section processing for each input frame, only histogram node access and last known bounding box access are placed in critical sections and they are independent. Since the processing involved in each critical section is much lighter than in the single unit case, the constraints put by the processing time on the system throughput are much weaker. Another advantage of design 2 is to make explicit the main data structures and the different components that are involved in the processing, and their relationships. The finer modularity of the system makes it more understandable and also more flexible in its implementation, maintenance and evolution. In the light of these observations, the CAMSHIFT unit of design 2 is worth a closer look.

3.3 Explicit CAMSHIFT architecture

As noted earlier, the core of CAMSHIFT is a feed-back loop in which the last known target bounding box is used to set the initial search location in the new input image. First, mean shift is used to compute the centroid of the 2D color probability distribution within the search window. second, the size and orientation of the target probability distribution are computed from the zeroth and second moments respectively. Finally, the last known target bounding box is updated, and will be used to compute the initial search location for the next frame.

Figure 5 shows an SAI design (*design 3*) in which the three CAMSHIFT stages (mean shift, moments and update) are explicit, and organized into a feed-back loop. Since the MeanShift and the Update cells

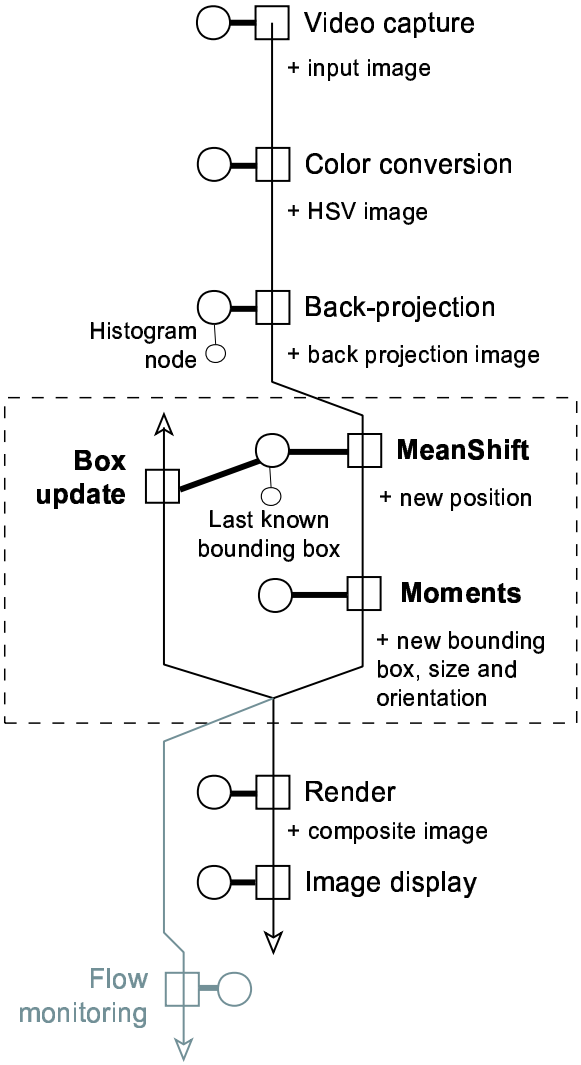

Figure 5: An explicit feedback loop design. The dashed delineates the subsystem functionally equivalent to the CAMSHIFT unit (step III) in figure 4. A flow monitoring unit allows to collect throughput and latency statistics with minimum impact on overall system performance.

both use the persistent last known bounding box, they are both connected to a common source that holds the bounding box node.

Because of the asynchrony of the processing, the target information used for initialization in the MeanShift 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 difference between the previous result used for input and the frame being processed will be more than the interval between two consecutive frames. The algorithm will then be applied in conditions similar to that of a system with lower throughput, and will perform consistently. However, thanks to the asynchronous parallelism, the higher overall throughput will be maintained for the system.

Another type of parallelism is illustrated in the branching of the stream to follow independent parallel paths. After coming out of the Moments cell, the stream follows a path to the Update cell, and another path through the rendering cell and finally to the display cell. While pulse-level multi-threading principally improves throughput, stream-level parallelism can have a major impact on latency. The result of the processing should be used as soon as possible for the application of interest (here visualization) and for update, in no arbitrarily imposed order. As long as computing resources (in a general sense) are available, and assuming fair scheduling, the model allows to achieve minimal latency.

Design 3 certainly gives more insight in the structure of the CAMSHIFT tracking system than the

black-box designs in either DirectShow or SAI. In fact, design 3 could (and, normally, should) be directly derived from a description of CAMSHIFT and its use for tracking. The system design graph itself gives an informational pictorial representation of the algorithm. Depending on the level of detail in which each element is described, the graph may be seen as a conceptual or logical level design. It is also to be used as a structural map of the code that implements it (physical level). Note that such a system would not be easily modeled with the same level of detail in a traditional dataflow approach. In particular, because they lack the concept of shared repositories, dataflow models make feed-back loops involving more than one processing unit ill-defined.

Design 3 makes good use of SAI modeling power and has a level of detail suitable to the communication and sharing of the design, and the efficient implementation of a corresponding system using OpenCV. The next section reports on the implementation of the three proposed designs and on their comparative performance.

4 Implementation

SAI designs constitute blueprints for implementation that map closely to the code. The three designs presented in section 3 were implemented using MFSM (Modular Flow Scheduling Middleware) [6], an open source architectural middleware for SAI. This section presents comparative numerical studies of coding effort and system performance. Complete source code and an accompanying tutorial are available online [7].

4.1 CAMSHIFT with MFSM

MFSM is an open source architectural middleware implementing the core elements of the SAI style. The project comprises extensive documentation, including user and reference guides, and several tutorials. Additionally, a number of software modules regroup specializations implementing specific algorithms or functionalities. The experiments related in this paper were implemented using MFSM, one system for each of the designs presented in section 3. The systems developed make use of existing modules as often as possible, to minimize the amount of coding required.

Software Line Of Code (SLOC) metrics help estimate code volume, and thus development effort. SLOC measurements were computed with the Code Count tool for C++ [1]. The physical SLOC definition is independent of the programming language syntax; the logical SLOC definitions involve language-specific syntax. Table 1 shows SLOC counts for the code implementing the CAMSHIFT DirectShow filter and for the code implementing analogous SAI elements (CAMSHIFT tracker cell and node in design 1). The simpler API offered by MFSM requires less than one third of the coding to achieve the same functionality. Even if, in both cases, most of the framework "wrapping" code can be generated automatically, a smaller code volume is always preferable, because simpler to understand. The demonstration applications using the filter and the module are quite different in their structure, and thus not directly comparable.

SLOC	Physical	Logical
DirectShow filter (OpenCV)	735	504
Tracker module (SAI design 1)	169	162

Table 1: SLOC metrics for CAMSHIFT tracker DirectShow filter and SAI module.

Table 2 shows SLOC counts for the different SAI designs. The numbers reported distinguish non-specific elements (MFSM core library and non-OpenCV related modules) from CAMSHIFT specific elements (module(s) and application graph). The tests do not include any user interface code. Only specific code must be actually written in order to implement the systems, non specific code is available as libraries or source files that can be included directly into a project.

As can be expected, as the design complexity increases, so does the amount of specific code to write. Yet, this amount only represents about one fifth of the total project SLOC count for design 3. As mentioned above, some of this code is "wrapper" code that could be generated automatically. The remaining code is directly related to the implementation of the specific algorithms of interest for the experiments or application. In the present case, this implementation is a mix of direct coding and use of specific OpenCV data structures

SLOC	Physical	Logical
Non specific	4059	2998
Design 1 sp.	277	263
Design 1 total	4336	3261
% specific	6%	8%
Design 2 sp.	734	579
Design 2 total	4793	3577
% specific	15%	16%
Design 3 sp.	963	794
Design 3 total	5022	3792
% specific	19%	21%

Table 2: SLOC metrics for the systems implementing the three SAI designs.

and functions. Note that hardly any code that is not directly relevant to a Computer Vision researcher was written: the MFSM core library, as well as video input and display, are examples of code that any researcher in the field should be able to take for granted.

Also note that the specific code developed for the CAMSHIFT demonstration would actually belong in an OpenCV encapsulation module, which would be directly reusable in subsequent projects.

4.2 Performance comparison

SAI's asynchronous parallel processing model allows to specify optimal designs in terms of throughput and latency. Critical questions in the context of this paper are: (1) how well MFSM implements the asynchronous parallel processing model; and, (2) what influence a system's design has on its throughput and latency. Efficiency and scalability of systems implemented with MFSM are supported by a large number of experiments reported for example in [10] and [9]. Multi-threading support enables real-time performance for media stream processing using current state-of-the art single or multi-processor machines, while allowing to achieve maximum throughput and minimizing latency.

A flow monitoring module (part of MFSM) implements a cell that collects throughput and latency statistics with minimum impact on overall system performance. For each design, measurements were taken as shown on figures 3, 4 and 5. The latency is measured as the time difference from the image time stamp (set at capture time), to the time when the detection data becomes available in the system. This latency measure does not take into account camera delay, which is a combination of the latency imposed by the capture rate, and hardware delays. Rendering and display of the composite image are also left out, to be consistent with the use of the tracking for PUI purposes.

Measurements were taken in similar conditions, with live video input at 320x240 and 640x480 pixels, for an effective throughput of 30 frames per second. For these experiments, the critical elements of the computer system used are: a dual Intel Pentium 4 at 2.2GHz CPU, a Logitech QuickCam Pro 4000 (USB) for the live video input, and an nVidia Quadro4 XGL graphics card for display. The latency as well as the processor load linked to the system showed no significant variation among the three tracking systems. For reference, the numerical values obtained in this particular experimental setting were: 320x240: <6ms latency and <20% processor load; 640x480: <30ms latency and <80% processor load.

These results suggest that, even for such a simple system, a significantly more complex design (in terms of architectural component instances involved) does not result in a significantly more computationally expensive nor less efficient software system. For more complex systems, full use of SAI's properties should result in significant performance improvements. The experimental results also bring supporting evidence that MFSM, although it is a proof-of-concept implementation yet to be fully optimized, is efficient enough for real-time video processing on modern computers.

5 Summary and perspectives

This paper presented design and implementation experiments for CAMSHIFT-based tracking systems using OpenCV and SAI. The reverse engineering of the OpenCV CAMSHIFT illustrated the intellectual and

practical value of a precise system representation that explicits the internal structure of the algorithms that it comprises as well as their relationships. An SAI design graph also represents a detailed blueprint for the coding of the system, whose implementation, using a corresponding architectural middleware such as MFSM, is then straightforward. A large part of the code can be directly imported as reusable modules, significantly reducing development effort. When used correctly, SAI confers properties to the system architectures designed in the style, that translate into efficiency and optimality properties for the software implementing those designs. The modularity of the resulting code and the availability of a detailed map (the design graph) facilitate system maintenance and evolution, code reuse in subsequent applications, as well as experiment reproducibility and objective comparative testing of various functionally similar algorithms in otherwise identical systems. Finally, even in the case of such a simple and straightforward system as the CAMSHIFT-based tracker, an explicit design provides much better understanding of the algorithm and its dynamics. Proof-of-concept system coding effort is kept to a low level, and performance of the resulting system is not impacted. In the case of complex systems, full use of SAI's properties should result in significant performance improvements.

Tracking is often thought of as "connecting the dots," an implicit off-line, trajectory-centered approach. The designs presented above (section 3) suggest a different approach. The systems operate under the assumption that there is exactly one target (encoded as the persistent "last known bounding box structure"). They rely on one single method of detecting evidence of the presence of this target in input images (detection and localization of a color distribution) to update their knowledge of the target. Tracking is not an algorithm, but an emergent property (or behavior) of the system performing detection and/or identification. A trajectory is a recording of the states of the target instance. Beyond the straightforward extension to multi-target tracking, these observations naturally point toward systems that will achieve robustness by combining many imperfect (but efficient) detection methods. SAI provides a formalism to not only describe the algorithms, but equally importantly how they interact in the system. In the design and implementation of such systems, a particularly challenging and exciting issue is the adaptation of current techniques for combining the contributions of individual detection algorithms (e.g. Bayesian networks). These composition concepts and challenges are central to the design of multi-modal PUI systems.

Acknowledgments

This work was supported in part by the Advanced Research and Development Activity of the U.S. Government under contract No. MDA-904-03-C-1786.

References

[1] CodeCount Tools. URL `http://sunset.usc.edu/research/CODECOUNT`.

[2] Intel Open Source Computer Vision Library. URL `http://www.intel.com/research/mrl/research/opencv/`.

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

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

[5] D. Comaniciu and P. Meer. Robust analysis of feature spaces: Color image segmentation. In *International Conference on Computer Vision and Pattern Recognition*, pages 750–755. San Juan, Puerto Rico, 1997.

[6] A. R.J. François. Modular Flow Scheduling Middleware. URL `http://mfsm.sourceForge.net`.

[7] A. R.J. François. CAMSHIFT Experiments. MFSM Tutorial, 2004. URL `http://mfsm.sourceForge.net/tutorials/Camshift.html`.

[8] A. R.J. François. A hybrid architectural style for distributed parallel processing of generic data streams. In *Proc. Int. Conf. Software Engineering.* Edinburgh, Scotland, UK, May 2004.

[9] A. R.J. François. Software Architecture for Computer Vision. In G. Medioni and S.B. Kang, editors, *Emerging Topics in Computer Vision*. Prentice Hall, 2004.

[10] A. R.J. François and G. G. Medioni. A modular software architecture for real-time video processing. In *IEEE International Workshop on Computer Vision Systems*, pages 35–49. Vancouver, B.C., Canada, July 2001.

[11] Microsoft. DirectX. URL `http://www.microsoft.com/directx`.

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

[13] M. Turk and G. Robertson. Perceptual user interfaces (introduction). *Communications of the ACM*, 43 (3):32–34, March 2000.