System and software architectures of distributed smart cameras — Source link

Chang Hong Lin, Marilyn Wolf, Xenefon Koutsoukos, Sandeep Neema ...+1 more authors

Institutions: National Taiwan University of Science and Technology, Georgia Institute of Technology, Vanderbilt University

Published on: 06 Apr 2010 - ACM Transactions in Embedded Computing Systems (ACM)

Topics: Systems architecture, Middleware, Smart camera, Gesture recognition and Software architecture

System and Software Architectures of Distributed Smart Cameras

CHANG HONG LIN
National Taiwan University of Science and Technology
MARILYN WOLF
Georgia Institute of Technology
and
XENEFON KOUTSOUKOS, SANDEEP NEEMA, and JANOS SZTIPANOVITS
Vanderbilt University

In this article, we describe a distributed, peer-to-peer gesture recognition system along with a software architecture modeling technique and authority control protocol for ubiquitous cameras. This system performs gesture recognition in real time by combining imagery from multiple cameras without using a central server. We propose a system architecture that uses a network of inexpensive cameras to perform in-network video processing. A methodology for transforming well-designed single-node algorithm to distributed system is also proposed. Applications for ubiquitous cameras can be modeled as the composition of a finite-state machine of the system, functional services, and middleware. A service-oriented software architecture is proposed to dynamically reconfigure services when system state changes. By exchanging data and control messages between neighboring sensors, each node can maintain broader view of the environment with integrated video-processing results. Our prototype system is built on Windows machines, and uses standard video cameras as sensors and local network as a communication channel.

Categories and Subject Descriptors: C.2.4 [**Computer-communication Networks**]: Distributed Systems—*distributed applications*; C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems; I.4.9 [**Image Processing and Computer Vision**]: Applications; I.6.5 [**Simulation and Modeling**]: Model Development—*modeling methodologies*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Distributed cameras, software architecture, smart camera

This work was supported by the National Science Foundation under grant ITR-0325119.
Authors' addresses: C. H. Lin, National Taiwan University of Science and Technology, M. Wolf, Georgia Institute of Technology, X. Koutsoukos, S. Neema, and J. Sztipanovits, Vanderbilt University.

DOI 10.1145/1721695.1721704 http://doi.acm.org/10.1145/1721695.1721704

ACM Reference Format:
Lin, C. H., Wolf, M., Koutsoukos, X., Neema, S. and Sztipanovits, J. 2010. System and software architectures of distributed smart cameras. ACM Trans. Embedd. Comput. Syst. 9, 4, Article 38 (March 2010), 30 pages.
DOI = 10.1145/1721695.1721704 http://doi.acm.org/10.1145/1721695.1721704

1. INTRODUCTION

A smart camera system is designed to both capture the video streams and use its own embedded processor to execute video-processing algorithms in a single package, which allows the users to not just watch video for activities of interest but also to use processing elements to automatically or semiautomatically identify the activities of interest. Smart cameras can perform various kinds of real-time video processing, including face, gesture and gait recognition, object tracking, and so on.

Surveillance, Human Computer Interface (HCI), and entertainment are the three major applications for automatic human action analysis, and a huge amount of camera systems have been introduced in the past decades. However, a system with only a single camera has a limited field of view. Furthermore, some objects inside the scene may be occluded from the point of view of a single camera. Physically distributed cameras are often used to aid in the analysis process, for example, to avoid occlusion.

In order to perform real-time video processing, distributed cameras require in-network processing power to handle computational tasks close to where the video is captured. A distributed smart camera system uses both physically distributed cameras and distributed algorithms to perform analysis. Instead of processing only the captured video streams, ubiquitous cameras can take advantage of communication with their neighbors to integrate the knowledge of the overall system without using centralized servers.

When analyzing video streams from multiple cameras, the fusion of the data from these cameras is a design challenge. Traditionally, multiple camera systems for video processing have relied on centralized servers: the captured data is sent to one central server (or perhaps a cluster of servers) for processing. Server-based video analysis systems simplify the synchronization and data-sharing problems. However, sending raw video data to the server(s) incurs some severe penalties, such as (i) a high-performance network is required to connect the camera nodes to the server, (ii) the network may consume a significant amount of energy that may be too high to be supported by systems with limited energy sources, and (iii) the transmitted video may be easily tampered with or disrupted in the server-based systems.

Many factors make the design of a real-time distributed system a challenging task. Human action analysis with a single camera is a hard problem, and its extension to multiple camera nodes is even harder. Other than distributed software, multiple camera systems also have to consider the communication between the cameras. In some systems, the cameras have to be registered and synchronized initially in order to make distributed algorithms realistic. Control authorities have to be passed among the ubiquitous sensors to perform distributed processing efficiently.

This article serves as the extension and summary of our previous work. The initial version of this system and the migration methodology was proposed in Lin et al. [2004]. The timing synchronization of our system is described in Lin and Wolf [2005]. Lin et al. [2006] and Kushwaha et al. [2006] used gesture recognition and vehicle tracking as examples to introduce the target-centric service-oriented framework. The control authentication protocol was discussed in Velipasalar et al. [2006]. This article also introduced an improved software framework, along with the protocols used in the middleware of the system.

Our major contributions of this article reside on the programming framework, software migration methodology, and novel peer-to-peer protocols for ubiquitous sensors. We propose a target-centric service-oriented framework, enabling the designers to develop ubiquitous sensor applications without dealing with the complexity and unpredicatability of underlying network dynamics. A target here refers to a person or an object moving around the field-of-view of the cameras. This framework provides designers with a higher level of network abstraction, allowing applications to be developed from the viewpoint of the targets. In order to reduce the effort in designing a distributed sensor system when well-defined single-node applications exist, it is desirable to reuse what the designers already have. A migration methodology to transform a well-defined single-node algorithm to a distributed sensor system is presented in this article. We introduce a gesture recognition architecture, smart camera, as a design example of ubiquitous sensors. The smart camera system includes video-processing algorithms, system modeling, and peer-to-peer protocols used for data exchange and authority control.

The next section summarizes previous work in smart cameras and distributed smart cameras. A service-oriented architecture for ubiquitous sensor system is proposed in Section 3, and its middleware is introduced in Section 4. Section 5 describes the migration methodology to reduce the efforts of designing distributed sensors. The smart camera is introduced in Section 6, as well as its system-modeling and authority control protocol. Finally, Section 7 concludes the article.

2. RELATED WORK

Although ubiquitous camera systems can be considered as a kind of sensor network, distributed systems of cameras post some new challenges when compared to the relatively low data rates of typical sensor networks. General sensor networks contain huge amounts of low-cost sensors with limited energy and computing power; while camera systems often have fewer nodes with much more energy and resource requirements.

Several architectures and algorithms for real-time camera systems have been proposed in the past decade. A great deal of work in computer vision has been dedicated to distributed camera, a term that means multiple, physically distributed cameras but does not imply anything about the algorithms used to process the images. This line of work has generally relied on centralized algorithms in which data from the cameras is sent to a server for analysis and fusion. Pentland [2000] surveyed several real-time video analysis efforts in detail. Besides algorithm development, hardware design is also an important issue for real-time systems. Bove's group at the MIT Media Lab proposed a data-flow model for real-time parallel media processing and built tiles of smart sensors [Watlington and Bove 1997; Bove and Mallet 2004]. The VSAM project was an early distributed smart camera system [Collins et al. 2001]. It consisted of a network of smart cameras that cooperated to perform tracking.

Pfinder was developed at the MIT Media Lab to detect and track people using 2D models with maximum a posteriori probability [Wren et al. 1997]. Cai and Aggarwal [1998] proposed a multicamera framework for tracking moving human beings in an indoor environment. Davis et al. [1999] developed a multiperspective video system for human action analysis and object detection and tracking [Mittal and Davis 2001]. Collins et al. [1999] built a video surveillance system using multiple cameras. Ozer and Wolf [2001] synthesized a 3D model of humans from two cameras approximately perpendicular with each other. Grieffenhagen et al. [2001] used omnidirectional cameras that use fisheye lenses or spherical mirrors to view a large area. Matsuyama and Ukita [2002] developed a real-time multicamera vision system in which the cameras are mechanically panned and zoomed. The Stanford multicamera array group uses a dense array of CMOS image sensors to capture multithousand frame-per-second videos [Wilburn et al. 2004]. Svoboda et al. [2005] developed a method to automatically calibrate three or more cameras using virtual 3D points generated by a laser pointer.

Fleck and Strasser [2005] use a network of cameras that perform tracking by handing over targets as they move between fields of view. Rinner's group at Graz University of Technology developed an embedded camera system for traffic monitoring and proposed methods for dynamic task allocation among a group of cameras [Bramberger et al. 2005; Bramberger et al. 2006]. Hengstler and Aghajan [2007] proposed an application-oriented approach to use a network of low-resolution image sensors on object tracking.

Some of the systems described can achieve real-time performance with centralized processing. However, sending raw video streams to centralized servers is neither efficient nor practical, especially when the transmission cost between camera nodes is measurable. Using shared memory or buses is unrealistic in real-life applications. A better choice would be to perform distributed computing involving the microprocessors inside or near the sensors with limited exchange of processed data. Velipasalar et al. [2006] used a peer-to-peer architecture to track targets; every camera with tracked every target in its field of view and shared its own track estimate with other cameras that could see the target.

As embedded camera systems become much more complicated, modeling and verifying such systems becomes harder. Ubiquitous distribution makes the design and verification of communication channels between sensor nodes even more difficult. Several groups have developed tools and environment for software architecture to model distributed systems. The model-integrated computing (MIC) is based on domain-specific models to analyze and test embedded software [Karsai et al. 2003]. The Virginia embedded systems toolkit (VEST) is a framework for component-based real-time systems [Stankovic et al. 2003]. The *CADENA* framework is an integrated environment for analyzing the CORBA component model (CCM) based systems [Hatcliff et al. 2003]. *Time Weaver* is a component-based framework that supports reusability of components across systems with parafunctional requirement [de Niz and Rajkumar 2003]. Holzmann and Joshi [2004] developed *SPIN* model checker to verify distributed software through message passing. In a similar framework, Rinner et al. [2007] developed embedded middleware on their distributed SmartCams.

3. SERVICE-ORIENTED ARCHITECTURE

In this section, we propose a software architecture for ubiquitous sensor systems. The environment used for this architecture is based on MIC by Sztipanovits and Karsai [1997], which uses domain-specific modeling languages to provide a flexible framework for embedded software development.

Traditional embedded systems perform fixed tasks as specified within the software, even when multitasking is used to perform multiple threads simultaneously. However, as systems become much more complicated, running all the possible services becomes impossible. Sensor nodes may have varying capabilities and provide different sets of functionalities. Naïvely executing all the services will not only increase the scheduling difficulty but also consume extra energy. Having a priori knowledge of what the system has to do at certain times helps to make the system more efficient. It is desirable to have systems not only provide complicated services, but also perform services at the requested time. A service-oriented architecture offers flexibility in the design of sensor applications, since it provides accepted standards for representing and packaging data, describing the functionality of services, and facilitating the search for available services that can be invoked to meet application requirements.

The service-oriented architecture is divided into three layers: application, middleware, and service. The application layer defines the objective of the combined services under it and can be a service itself for larger applications. As illustrated in Figure 1, we use tracking as the example of objective application.

The service layer contains all the services offered to the application. A service is a software architecture that performs certain tasks. A service can be enabled or disabled on-demand based on the current state of the system. Each sensor node may provide several different services. The required services are specified for each system state and are dynamically bound at runtime. The services not only refer to different types of functions, but also to different algorithms with the same functionality. Services are modular and autonomous, which permits them to be dynamically composed into complete applications. At a system state, the sensor node may require a series of functional services. Depending on the environmental settings, the nodes may choose a different service with

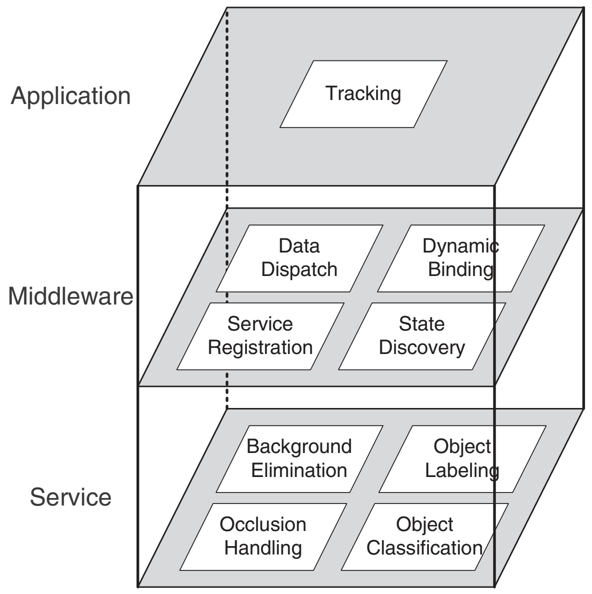


Fig. 1. Service-oriented software architecture example—tracking application.

similar functionalities. For example, a sensor node may require a background subtraction service followed by a tracking service, and due to the lighting condition, the sensor node may use a different background subtraction algorithm. As stated in Figure 1, background elimination, occlusion handling, object labeling, and object classification are several possible services for the tracking application.

The middleware layer defines the interfaces of the services to handle dynamic binding and message passing of the required services for a system state. Each service should provide the information of input and output channels of the functional block as well as the service attributes to the middleware, and the middleware can then register the services and dispatch the data messages between them. The services with similar functionalities should have the same set of input and output channels, with possibly different service attributes. Using a service requires knowing only its name and interfaces. When a system state changes, the middleware would discover the change and then dynamically reconfigure the required system services.

3.1 System Modeling Phases

The system software modeling using a service-oriented architecture consists of three major phases: At the application layer, construction of the system operation finite-state machine (FSM); at the service layer, definition of the possible services, either atomic or composite; and at the middleware layer, determination of the mapping between the system states and provided services. Figure 2(a) illustrates the three phases in the system modeling.

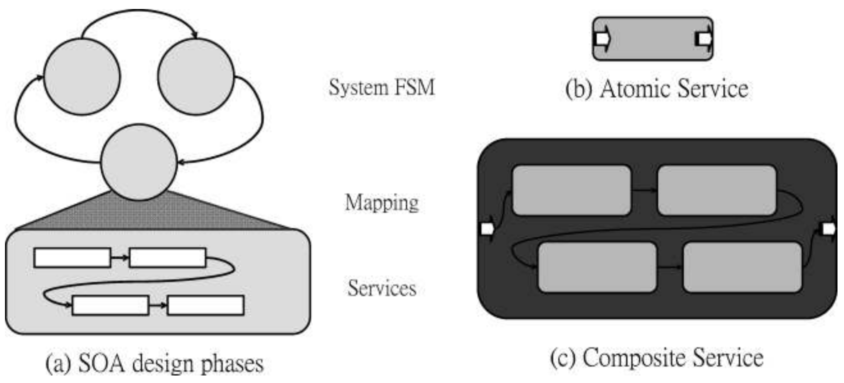


Fig. 2. (a) The three phases for software modeling using a service-oriented architecture and the application services: (b) atomic services and (c) composite services.

The application services in service-oriented architecture modeling can be divided into two categories: atomic services and composite services. Atomic services are the basic building blocks for a certain sensor system that can provide specific functional tasks. Each application service has a unique name that can be indicated by the middleware, and their input and output ports are specified in advance, as illustrated in Figure 2(b). Services with similar functionalities have the same set of interfaces that can be exchangeable with each other. The atomic services can be wired together to form more complex behaviors. The aggregated atomic services then form a composite service, as shown in Figure 2(c). Composite services are formed by specifying data paths through atomic services.

There is no unique way to construct a system FSM. One possible solution for ubiquitous sensors is target-centric modeling. For each target object that enters the scene, the system creates a target FSM, changes the FSM state as the target moves around or when noticed by the services, and destroys the FSM when the target leaves the environment. Hierarchically, a well-defined application FSM can also serve as an atomic service of higher-level applications by using a service wrapper around itself. In a target-centric application, a target is a unique logical entity corresponding to a physical phenomenon being monitored by the camera system. The target is required to reside on only a single node at a time; it migrates between nodes as it follows its real-world counterpart. The system application is then driven by the targets; that is, its behavior reflects the target's current state. This effectively transfers the ownership of common tasks such as computation and communication from the individual nodes to the target itself, providing flexibility and efficiency both at design time and runtime.

3.2 Metamodeling of Service-Oriented Architecture

The first step for modeling a distributed system using a service-oriented architecture is to configure the environment. The configuration step is itself a form of modeling, the modeling of a modeling process, and is usually called *meta-modeling*. The output of the metamodeling process is a compiled set of rules, the paradigm, for a specific application domain. The metamodeling process only has to be performed once for a certain application domain. Once the metamodel is constructed, the designers can then use it to model different applications in the defined domain.

As in designing an embedded system, the first step is to construct the specification of the modeling application. The designers then perform the metamodeling process based on the specification. The major concept for metamodeling process is to determine the entities used by the application model and the relations between the entities. The information used to identify and qualify certain entities and relations will be assigned to them as attributes. In short, metamodeling is the mapping of specification concepts onto entities, relations, and attributes.

Target-centric modeling can be used to model the FSM for distributed cameras using the service-oriented architecture modeling. When a target object enters the fields-of-view (FOVs) of the cameras, the system will spawn a new FSM to model the processing performed on the target. The system FSM changes its state as the target moves around the view of the cameras. Atomic services are provisioned on sensor nodes, and composite services are created from atomic services. Services are then dynamically bound according to the current state of the system.

Generic Modeling Environment (GME) is used as the design environment to construct the metamodeling for distributed camera systems. The metamodel of the target-centric modeling is divided into four aspects: target object, service graph, control flow, and data flow, as shown in Figures 3 and 4. In the target object aspect, the states of the system FSM is defined, as illustrated in Figure 3(a). Each state is refined as a service graph, and the transition between the states is triggered by guards involving target attributes, which are stored as variables. As shown in Figure 4, the service graph for a given system state can be further broken into control flow and data flow graphs. Tasks are used to coordinate the services, including services in both the control flow and data flow. The metamodel for the data flow graphs is illustrated in Figure 3(b), which consists of a starting node, end node, and intermediate nodes. The execution flow may also be determined by conditional guards, as stated in the control flow graphs. Figure 3(c) shows the metamodel for the control flow graphs. Each control flow graph contains a single start node and a single end node, and the control flow may be conditional, fork into concurrent processes, or join with other concurrent processes.

3.3 Target-Centric Programming Framework

The basic idea of target-centric programming is to provide designers with a higher level of system abstraction, allowing applications to be developed from the viewpoint of the target object. In target-centric modeling, targets are objects representing physical phenomena being monitored by the system.

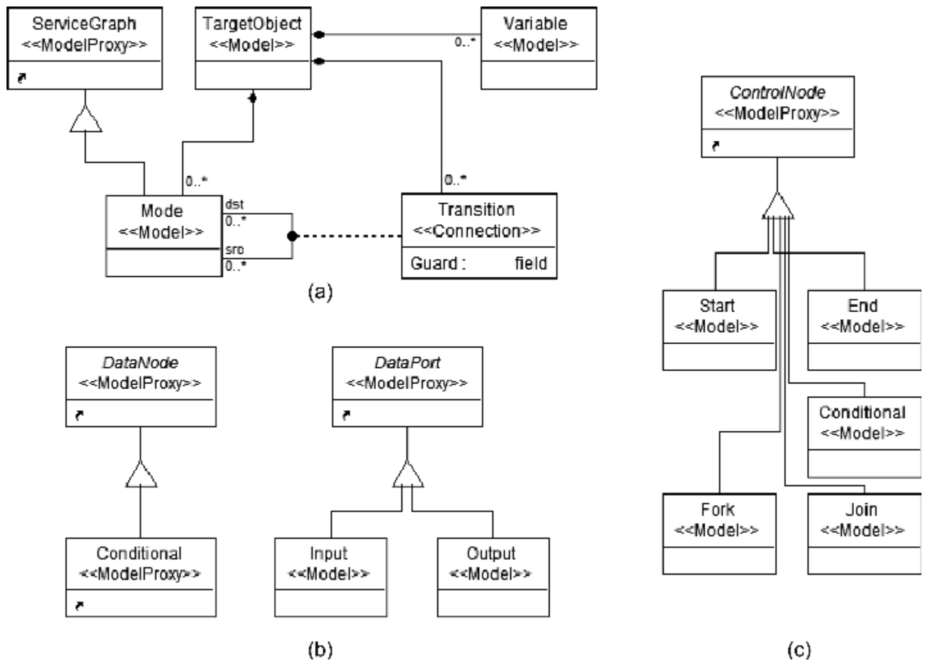


Fig. 3. The (a) target object, (b) data flow, and (c) control flow aspects in metamodeling for distributed cameras using GME.

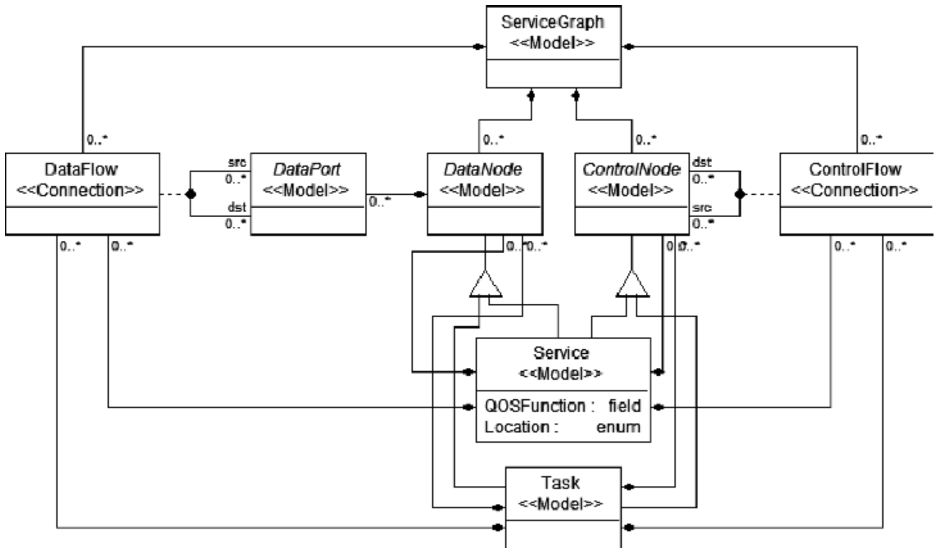


Fig. 4. The service graph aspect in metamodeling for distributed cameras using GME.

Target-centric programming can be used for a variety of applications in ubiquitous sensor systems, such as distributed gesture recognition, fire detection and monitoring, and mobile vehicle tracking. These algorithms are typically resource intensive and require collaboration among several sensor nodes.

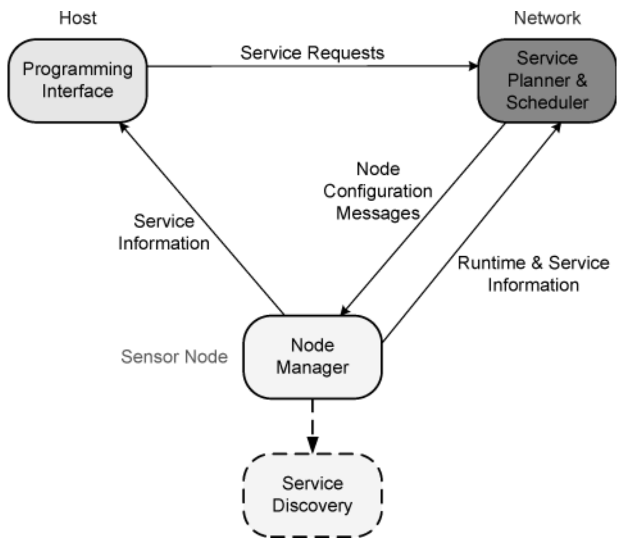


Fig. 5. The basic architecture of the target-centric programming framework.

While monitoring the target object, the tracking agent must simultaneously locate various remote image-processing service components and forward processed image data. After video processing, other services may use the results to perform any necessary actions. To develop such a programming paradigm, the target-centric programming framework uses a service-oriented architecture to perform behavioral decomposition. Each activity is implemented as a separate service, which is modular and autonomous, and has well-defined interfaces. These properties permit services to be dynamically composed into complete applications. Since most ubiquitous applications consist of multiple services on multiple nodes, planning, scheduling, and service discovery mechanisms are built into the middleware of the target-centric service-oriented framework.

The basic architecture of the target-centric programming framework is shown in Figure 5, which consists of three major components, namely the programming interface, service planner and scheduler, and node manager.

The *programming interface* is located at the host machines and is responsible for injecting service requests into the system. The end users may create applications using the services registered in the service repository, which is acquired from the ubiquitous sensor nodes. The service repository keeps a registry of all the services discovered in the entire system. The service request outputs are sent to the service planner and scheduler.

The *service planner and scheduler* takes the service requests as its input and generates a service graph of constituent services listed in the service repository. A service request is expanded into constituent services until saturation, which means no more services need to be expanded. The scheduler then takes the service graph to schedule services at the sensor nodes that provide target-sensing functionalities while satisfying system specification and constraints. The scheduler requires runtime information of the services from the sensor nodes as well as the properties of the nodes such as location, remaining power, and so on. The output of the scheduler consists of node configuration messages to let the sensor nodes reallocate the provided services. For a centralized multicamera system, this component may be located at the server(s). For distributed multicamera systems, this component has to be written in the form of concurrent processes executing on distributed nodes for peer-to-peer cameras.

The *node manager* is located in each sensor node and is responsible for configuring the node services based on the configuration messages from the service planner and scheduler, initiating service discovery, and managing the runtime service and resource information. The node manager initiates service discovery to locate the services to execute on the target objects following the current service. A service discovery protocol is needed in the sensor nodes to allocate the required services in a distributed fashion.

4. MIDDLEWARE FOR UBIQUITOUS SENSOR SYSTEMS

As stated in the previous section, the service-oriented architecture models the ubiquitous sensor systems as system FSMs. The operations provided by the systems are decomposed into basic functional blocks, which are called *services*. The services executed at each node are determined according to the current state and system constraints. A middleware is needed to register the services and dynamically bind the required services in order to perform desired operations at the sensor nodes. In this section, a middleware suitable for modeling the service-oriented architecture is proposed.

4.1 Middleware Programming Model

In the service-oriented architecture, the target contains one or more service graphs whose constituent services provide the application with its desired functionality. The service graphs for a target are assumed to be known a priori and contain information necessary for locating services across the network. Specifically, a service graph shall contain a set of services, a set of bindings, and a set of constraints. A service consists of a service identification (ID) and I/O port IDs; a binding is a connection between two service ports, and a constraint is a restrictive attribute relating one or more services. Figure 6 shows the service graph for a vehicle tracking system. The service requires vehicle classification and position as its inputs. The vehicle classification service provides classification and position but requires a logical object for the target vehicle, while the vehicle is sensed by a vehicle-sensing service. The logical object is created by the target creation service. A constraint of the system is also listed in the figure: the position of the three sensors cannot be colinear.

Services are resources capable of performing tasks that form a coherent functionality. They have a well-defined interface, which allows them to be described, published, discovered, and invoked over the system. Each service has its own input and output ports to communicate with other services, and the communication follows the globally asynchronous, locally synchronous (GALS) model

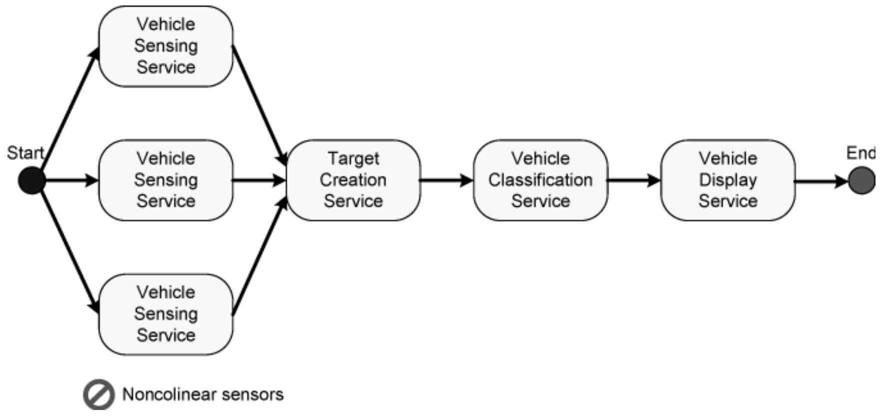


Fig. 6. A service graph example: vehicle tracking system.

of computation [Cheong et al. 2003]. GALS guarantees that communication between services will occur asynchronously, while intraservice communication will exhibit synchronous behavior.

4.2 Service Discovery, Composition and Constraints

Before a target can start executing a service graph, a service discovery protocol (SDP) is invoked to determine which nodes in the network provide which services. The SDP maintains two service repositories, a local service repository (LSR), which catalogs the application services running locally, and a discovered service repository (DSR), which catalogs remote application services that have been discovered in the past. The service discovery algorithm, as described in Figure 7(a), receives as input a service ID, which, if not present in either service repository, will prompt the SDP to broadcast a service request to other nodes in the network. The outgoing service discovery message contains the ID of the requested service and the node ID of the sender and is broadcast throughout the network. Nodes providing the requested service will send a service discovery reply message, which includes the vital information for nodes, such as physical location and remaining power. The SDP caches the provider node ID in the DSR and forwards the message to the service scheduler.

It is the service scheduler's responsibility to produce a set of service and service provider pairs that satisfy the constraints specified in the service graph. These services are then bound and eventually invoked. Figure 7(b) outlines the behavior of the service scheduler. The ID of each service in the service graph is passed to the service discovery protocol. Since the same service may reside in multiple nodes across the network, the scheduler has to expect multiple replies. As replies arrive, the scheduler checks to see if any atomic service graph constraints are satisfied and saves the node information accordingly. Compositional constraint satisfaction commences after all the replies have been received. Finally, the connections between the services in the service graph are examined, and a service binding message is created for each connection. The message simply contains the service and node identifications of the connection

```
Input: Service ID
search the Local Service Repository (LSR)
if Service ID is found in LSR then
    send local Service Info to Service Planner
end if
search Discovered Services Repository (DSR)
if Service ID is NOT found in DSR then
    compose Service Discovery Message
    broadcast Service Discovery Message
    receive Service Discovery Reply Message
    record service provider node ID in DSR
end if
send remote Service Info to Service Planner
```

(a)

```
Input: Service Graph $\mathcal{G}$
parse $\mathcal{G}$ into sets of Services, Connections and Constraints
for all $S \in$ Services do
    send $S$ to Service Discovery Protocol
end for
receive Service Discovery Reply from SDP
if node satisfies Atomic Constraints then
    cache node information
end if
do Compositional Constraint Satisfaction
for all $C \in$ Connections do
    create a Service Binding Message
    send Service Binding Message to service provider node
end for
```

(b)

```
$\dot{D}_i = D_i$
for all $v_i \in \dot{D}_i$ do
    if !satisfy($C_i$, $v_i$) then
        $\dot{D}_i = \dot{D}_i - v_i$
    end if
end for
```

(c)

```
for all $C_i \in \mathcal{C}$ do
    $\dot{\mathcal{D}}$ = prune_design_space($C_i$, $\mathcal{D}$)
end for
okay = FALSE
while !okay do
    $sol = \{(v_{index1}, v_{index2}, \dots, v_{indexn}) | \forall i\ v_{indexi} \in \dot{D}_i\}$
    okay = TRUE
    for all $C_j \in \mathcal{C}$ do
        if !satisfy($C_j$, sol) then
            okay = FALSE
            backtrack()
        end if
    end for
end while
```

(d)

Fig. 7. The pseudocode for the (a) service discovery protocol (SDP); (b) service scheduler; (c) atomic and (d) composition constraint satisfaction.

source, as well as the service and node identifications of the destination. The message is sent to the connection source node so that it may properly direct the output of its service to the input of the service specified by the connection destination.

Once the target object has finished initialization, the service graph can be executed. This involves the invocation of the source services in the service graph. Depending on the nature of the target, the service graph may be executed periodically, in which case the source services are invoked at a predetermined rate. Because each application service invokes the next, the service graph will execute to completion without the need for any type of centralized control.

Service graph instantiation can be modeled as a constraint satisfaction problem, where services in the abstract service graph are the constraint variables, and the nodes that provide a particular service constitute the domain of the corresponding variable. A finite constraint satisfaction problem (CSP) $\mathsf{P} = (\mathsf{X}, \mathsf{D}, \mathsf{C})$ is defined as a set of n variables $\mathsf{X} = x_1, \dots, x_n$, a set of finite domains $\mathsf{D} = D_1, \dots, D_n$, where D_i is the set of possible values for variable x_i, and a set of constraints regarding the variables $\mathsf{C} = C_1, \dots, C_m$. A constraint C_i

is defined on a set of variables $(x_{i_1}, \ldots, x_{i_j})$ by a subset of the Cartesian product $D_{i_1} \times ... \times D_{i_j}$. A solution is an assignment of values to all the variables, which satisfy all the constraints.

A design space for a constraint satisfaction problem is the set of all the possible tuples of constraint variables. Formally, $\mathsf{D} = \{(v_1, v_2, \ldots, v_n) | v_1 \in D_1, v_2 \in D_2, \ldots, v_n \in D_n\}$. A consistent design space is defined such that given any instantiation I of any $n-1$ variables satisfying all the constraints among those variables, there exists an instantiation of other variables that satisfies all the constraints involving variables in I. Note that a consistent design space does not mean that any design solution is a feasible one, but for any solution for a variable, there exist assignments for other variables, such that the design solution is feasible.

The main idea behind constraint satisfaction is to prune the design space as much as possible for different types of constraints, followed by backtracking until a feasible solution is found. Atomic constraints are straightforward to satisfy, as shown in Figure 7(c). Because each atomic constraint is defined on a single variable, pruning the domain of that variable will leave the domain consistent and hence satisfy the constraint. Figure 7(d) outlines the underlying process of compositional constraint satisfaction. In general, higher-level complex constraints are more difficult and demanding to satisfy. However, such constraints can be transformed into lower-level, simple constraints that provide the desired result, while minimizing the power and resources expended in satisfying the constraint [Guibas 2002].

4.3 Middleware Summary

Middleware provides a layer of network abstraction, shielding the application programmer from the low-level complexities of sensor node operation, such as resource management and communication. It gracefully handles the decomposition of desired application behavior to produce node-level executable code for a target-centric service-oriented multicamera application.

The middleware includes a node manager, a service discovery protocol, and a service scheduler and provides support to the target object and application services. The node manager is responsible for message routing between services, including both local and remote messages. The header of the message handled by the node manager consists of the node and service identifications for both source and destination services and the message type. The node manager examines the header and determines the appropriate destination for the message.

Three key types of messages are handled by the node manager. The service discovery messages come from the neighboring nodes inquiring if specific services are available. The node manager passes messages of this type to the local service discovery protocol. An incoming service binding message indicates that a local service has been registered for use by a target object and includes information of where to send its output data when the service completes. A service access message is a request to run a local service and may also contain input data. The node manager invokes the specified service and passes the data

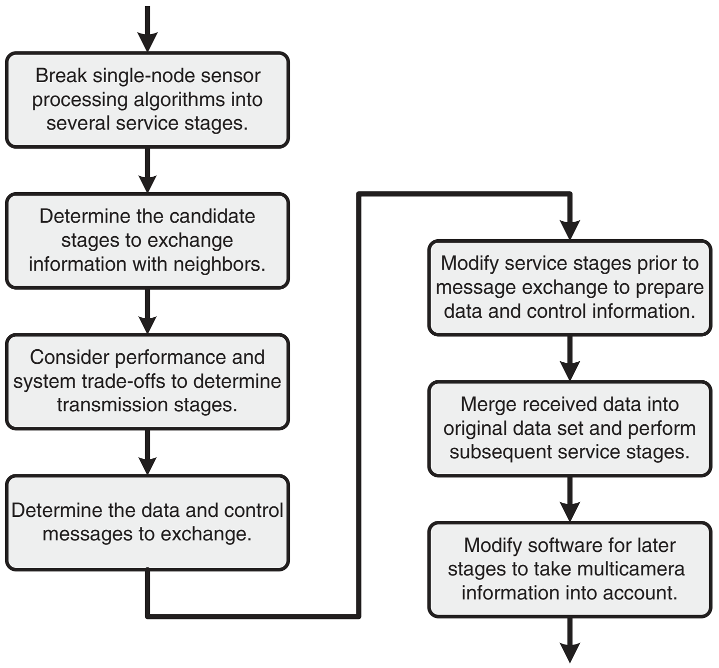

Break single-node sensor processing algorithms into several service stages.
Determine the candidate stages to exchange information with neighbors.
Consider performance and system trade-offs to determine transmission stages.
Determine the data and control messages to exchange.
Modify service stages prior to message exchange to prepare data and control information.
Merge received data into original data set and perform subsequent service stages.
Modify software for later stages to take multicamera information into account.

Fig. 8. The block diagram of the proposed migration methodology.

between service pairs. The service discovery protocol and the service scheduler are wired into the node manager and function, as described in Figure 7.

5. MIGRATION METHODOLOGY

The system architecture for distributed sensor applications has to be carefully designed in order to fulfill the system requirements. Even though there are no universal rules to make the design decisions, sensor geometry, image-processing software, communication channel, cost budget, power consumption, and battery life are important considerations. The total system cost has to be within budget, including equipment, development, installation, and maintenance costs. The sensors have to provide the required resolution and sensitivity. The computing devices have to be able to handle the processing, control, and communication in real time. The FOVs of the sensors have to cover the area of interest specified in the application requirements. The underlying network has to be able to provide sufficient bandwidth for the communication. The batteries have to be able to support the camera geometry set-up. The resulting system architecture may differ from an application to another; as long as the architecture can fulfill the system requirement and can perform the desired application efficiently, it would be an appropriate system architecture.

5.1 Migration from a Single Sensor Node

The proposed migration methodology is summarized in Figure 8 and can be used for ubiquitous video analysis systems. To change a single-node application into a ubiquitous sensor system, the first step is to determine which parts of the single-node system can be directly inherited and which parts cannot. The sensor nodes have to exchange information with the neighboring nodes in order to get an overall view of the entire system. The single-camera programs are divided into several service stages based on the software architecture, and the results from the service stages are the candidate data to be transferred to the neighboring sensor nodes. Each distributed sensor node exchanges data messages and uses the captured and processed video streams along with the received data to perform subsequent processing. Basically, the distributed programs run in each node can perform most of the single-node operations with additional multisensor controls, including preparing the messages to exchange. However, the information that needs to be transferred has to be determined in earlier processing stages, and the received data messages have to be taken into account in later processing stages.

After the service stage to exchange the information is determined, the designers have to decide what types of data is passed to the neighbors and what types is processed only locally in the current node. The data passed depend on the application itself, performance requirement, communication cost, and other application-dependent issues. The decision can only be made after taking into account all of the system trade-offs. The single-node software can then be modified to collect the information that needs to be transferred and exchange the messages through the communication channel. In order to minimize the changes made to the software, after the sensor nodes receive messages from their neighbors, each node merges the received data with its own dataset, if possible. The software for service stages after data exchange is then modified to suit distributed sensor applications.

5.2 Control Authority Determination

Besides the image data exchanged between the camera nodes, some applications might want to exchange control signals for many other purposes. Some applications may need the nodes to exchange ownership of the objects inside the scene to determine which sensor node performs most of the high-level processing for a certain object. Some sensor nodes may want to notify their neighbors that certain objects are moving toward their FOVs, or send out the characteristics of certain objects to their neighbors to let the neighbors identify them. The control signals can also be used to perform periodic timing synchronization or spatial calibration. Depending on the application, the distributed sensor system may use a separated channel for control signals or can embed the control signals within the data packages.

6. SMART CAMERA: GESTURE RECOGNITION SYSTEM

Our distributed gesture recognition system is based upon the single-node smart camera system developed by Ozer et al. [2000] and which has undergone several rounds of improvements [Ozer et al. 2001, 2005; Wolf et al. 2002; Lin et al. 2004].

The major objective for the smart camera system is to perform real-time gesture recognition in an embedded system. Gesture recognition is a particularly

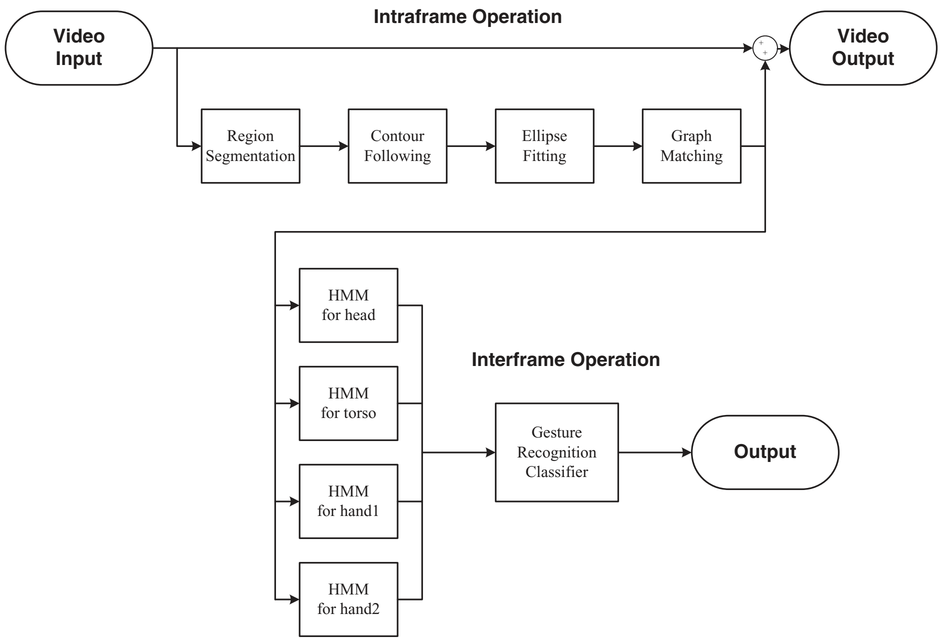


Fig. 9. The algorithm architecture of the single-node smart camera system.

challenging application for peer-to-peer smart cameras because the subject may span several cameras. Unlike tracking, in which a relatively small target can be handed off relatively cleanly, gesture recognition requires a more detailed model of the subject. Our system builds a unified model for a subject, even if different parts of the subject are visible to different cameras. It does so without transmitting raw imagery between the cameras—rather, it builds an intermediate model of the image that is shared for further processing on a remote node.

The software architecture of a single-node smart camera is illustrated in Figure 9, which consists of the intra and interframe processing parts. The intraframe part performs human body detection and extracts the abstract graph representation parameters, while the interframe part determines the movements and gestures of the people within the scene.

6.1 Intraframe Video Processing

The input video streams are either captured as raw video or decompressed to create a series of $M \times N$ bitmap images. The smart camera system then applies four algorithm stages to the bitmap frames in order to detect and identify the body parts in the intraframe processing, which includes region segmentation, contour following, ellipse fitting, and graph matching.

In the first stage, region segmentation, the system performs two video-processing tasks on the input video streams. The first step is background elimination, which determines the foreground regions by selecting parts of the frame that display motion. Our current implementation uses a simple background elimination algorithm—the absolute difference between the input frame and the background is calculated for each pixel, and the pixels with the difference larger than a certain threshold are considered as foreground regions. The threshold chosen is determined according to several environment conditions, including lighting condition or background colors. A more sophisticated background elimination algorithm, such as mixture-of-Gaussians, could also be used. The background elimination algorithm only distinguishes the foreground regions from the background model; however, for understanding gestures, it is also important to distinguish among the different body parts. The second step in the region segmentation stage is skin-tone detection. A skin color detection algorithm is used to separate head and hands from torso in the smart camera system using a YUV skin-tone color model with chrominance values downsampled by a factor of 2. This stage hierarchically segments the input frame into skin-tone and non–skin-tone regions by extracting foreground regions adjacent to the detected skin areas and combining these segments.

In the second stage, a contour-following algorithm is used to extract the boundaries of foreground regions, which involves linking the separated groups of pixels into contours that geometrically define the boundaries. A 3×3 filter is used to walk along the edge of the components in any of the eight different directions in order to generate the boundaries for the detected regions. The detected boundary pixels are then stored into an array indicating the pixel locations, region number, and skin-tone indicator.

In the third stage, the smart camera system fits an ellipse to each detected region as a model of that part of the body. The subject may be occluded by some background objects within the scene or even the limbs or clothing of the person himself/herself. A two-dimensional (2D) approximation of the body parts by fitting ellipses with shape-preserving deformations provides results that are more satisfactory. Hence, instead of the region pixels, parametric surface approximations are used to compute shape descriptors for segments such as area, compactness (circularity), weak perspective invariants, and spatial relationships. The ellipse parameters used to represent a foreground region include the center of gravity, two semiaxis, and ellipse orientation.

The last stage of the intraframe processing, graph matching, maps the extracted regions modeled with ellipse parameters to a human body part in a graphical representation. A human is a complex object formed by several simple visual parts such as head, torso, hands, and so on. The learning of the shape of the object of interest is then related to the learning of the organization of simple visual forms that make up the object of interest with different attributes and spatial relationship among themselves. Although graph matching is widely used for the representation of complex objects and scenes [Ballard and Brown 1982; Caelli and Bischof 1997], it faces problems due to the dependency on the region segmentation results.

According to the different articulated movements and clothing, the extracted features for human body detection may also differ. A piecewise quadratic Bayesian classifier uses the ellipses parameters to compute feature vectors consisting of binary and unary attributes of the human bodies. The unary features for the human bodies include compactness, eccentricity, and hair and skin colors. The binary features for the human bodies include ratio of areas, relative position and orientation, and adjacency information between nodes with overlapping boundaries. Object detection is achieved by matching the relational graphs of objects with the reference model. The aspect graph of the reference object is formed according to the segmentation results of the training images. Multidimensional Bayes classification is used to determine the body parts under the assumption that the unary and binary features belonging to the corresponding body parts are Gaussian distributed.

6.2 Inter-Frame Video Processing

The interframe processing component, which can be adapted to different applications, compares the motion pattern of each body part in a set of frames to the patterns of known postures and gestures and then uses several hidden Markov models (HMMs) in parallel to evaluate the person's overall activities. The motion pattern of a body part is described as a spatiotemporal sequence of feature vectors. We use discrete HMMs that can generate eight directional codewords that check the up, down, left, right, and circular movement of each body part. The motion patterns of all the body parts are combined to classify the gesture of the target.

Human actions often involve a complex series of movements. We, therefore, combine each body part's motion pattern with the one immediately following it to generate a new pattern. Using dynamic programming, we calculate the probabilities for the original and combined patterns to identify what the person is doing. We observe that different activity patterns can have overlapping periods (same or similar patterns for a period) for some body parts. Hence, the detection of the start and end times of activities is crucial. Gaps between gestures or activities help indicate the beginning and end of discrete actions. A quadratic Mahalanobis distance classifier combines HMM output with different weights to generate the reference models for various gestures.

6.3 System Migration and Partitioning

Our main purpose is to perform gesture recognition in the distributed camera framework. Each camera node has its own processing elements and uses the captured video streams and the received messages from its neighbors to find out human activity patterns inside its own FOV. Neighboring cameras are set up with slightly overlapped regions and approximately parallel FOVs. The camera nodes are registered in advance at the background plane. Then, the target people can freely move around the area close to the background plane, where the overlapped regions can be approximately modeled by the FOV lines.

For the distributed gesture recognition system with several approximately parallel cameras, if a target person stays entirely in the FOV of a certain camera, the single-camera system described can successfully recognize the movement of the target. However, problems will arise when the target person is moving around the boundaries of the cameras. Each camera covers a part of

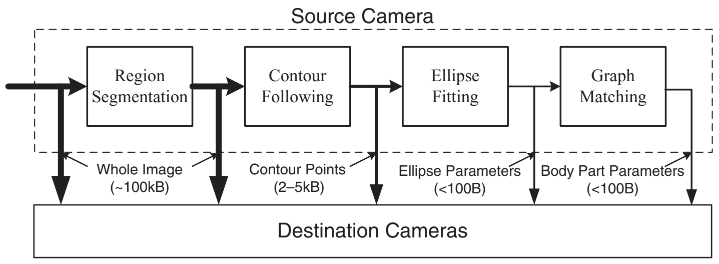


Fig. 10. The transmitted data size for different candidate stages of our distributed gesture recognition system.

the target person, and it is possible that neither of the cameras covers all the body parts of the target person. Therefore, these cameras have to exchange information on the target in order to obtain correct recognition results.

Before exchanging messages, the sensor nodes execute regular single-camera algorithms and prepare data and control messages for transmission. If no object is detected around the overlapped regions, the sensor nodes only exchange necessary control messages. On the other hand, if objects exist around the overlapped regions, the sensor nodes exchange prepared data and control signals with their neighbors. After receiving messages from the neighboring sensors, the received data information is integrated into the locally detected data set, while the control messages are used to determine which node may go on to process the target person. Only the chosen camera will go on to execute the successive processing stages.

In the gesture recognition system, the outputs of intraframe services are candidates for data transmission. Since a person may stay around the overlapped region of a camera pair, some body parts may be located outside its own captured image. Both region segmentation and contour-following service stages use pixel-based algorithms. Their inputs contain the whole image, which is too large to be sent. Thus, ellipse fitting would use the contour points to generate abstract ellipse parameters and fetch them into the matching procedure. The data size for the contour points is several KB, while the other possibilities, ellipse parameters, and matched body part with HMMs coefficients, often cost less than 100 bytes. When the network bandwidth is large, the contour points are good candidates for transmission; while the ellipse parameters and matched body parts are better choices when the network bandwidth is very limited. The data size transmitted from different stages is illustrated in Figure 10.

In order to recognize the gestures of a person at the boundaries, body part information has to be passed between sensor pairs with overlapped FOVs. Since the size of a human body is bounded, there is no need to deliver all the data (contours, ellipses, or body parts) for the detected regions in the FOV to the nearby cameras, but only the ones that lie inside or close to the overlapped region need to be transferred.

When the camera nodes process the captured video streams in contour-generating, ellipse-fitting, or graph-matching phases, besides the normal operations performed by the single-camera system, distributed sensors also check if the contour, ellipse, or body part lies in or near the overlapped region. The cameras would then only collect data of the detected foreground regions around the overlapped area and send them to the nearby cameras. Since the orientation of the cameras is known once the cameras are registered, the transferred information can be adjusted based on the coordination of the destination camera.

When a camera gets data from its neighbors, the received contour points, ellipse parameters, or body parts are combined with the captured matching data of its own to form a new dataset. The new data set contains three types of data: (i) the ones far away from the overlapped region, (ii) the ones outside its field-of-view, and (iii) the ones in the overlapped region. Basically, data type (i) and (ii) can be derived from the captured image and the received data directly; however, data type (iii) has to be handled more carefully. The matching regions within the captured image and the received data are obtained first. Then the one containing more pixels are considered as the dominant contour, ellipse, or body part.

The major part of the object of interest is then used to determine the ownership of the object, for example, *head* part can be used to determine the ownership of a human body. The camera with the dominant major part goes on executing higher-level algorithms on a certain object, while the other sensors discard all the body parts of the object, since they would be traced by other cameras.

6.4 Models of the Distributed Gesture Recognition System

Suppose target-centric modeling is used to model the ubiquitous sensor system. As each target person enters the FOVs of the cameras, the system spawns a new system finite-state automaton to handle the services activated by the target. For each system state, the sensor nodes perform a different set of services to fulfill the requirements of the application. Here, we assume the system has two slightly overlapped cameras with approximately parallel FOVs. The two cameras are initially registered in the background plane using the FOV line recovery method proposed by Velipasalar and Wolf [2004], and the targeted persons can freely move around the area close to the background plane.

The target-centric system finite-state automaton is shown in Figure 11(a), which consists of three states determined by the position of the targeted person. The system state automaton is created when a targeted person first walks into the scene, and as the person moves around, the system state may change to three possible states: target recognized by camera 1, target recognized by camera 2, and target recognized by both cameras. The system automaton is destroyed once the targeted person moves out of the views of the cameras. The transition between different states are determined by the guards on system variables such as time, four sets of body part parameters, and control tokens.

In different system states, the gesture recognition system provides a different set of services. When the targeted person is only located inside one

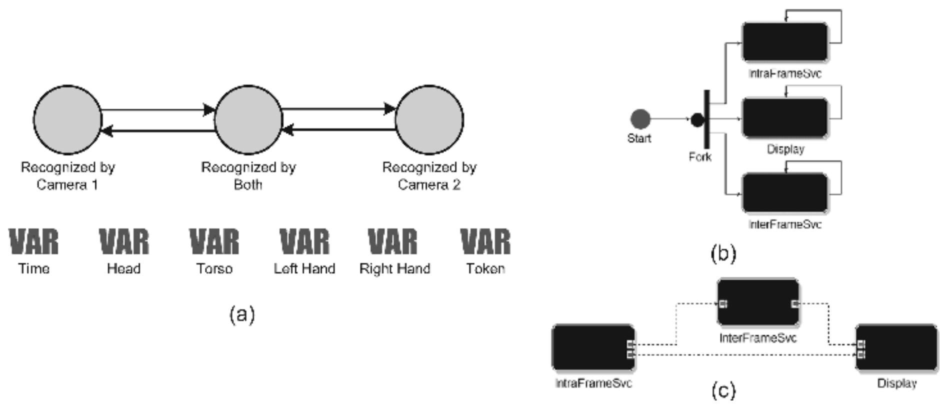


Fig. 11. The (a) system finite-state automaton, (b) control flow, and (c) data flow graphs of the distributed gesture recognition system with two cameras.

of the camera's FOV, only the camera with the target needs to perform gesture recognition, and the other cameras can stay in the low-power mode and wake up again when the system changes to the target in the overlapped region (recognized by both cameras) state. The services performed in these states are the same as single-camera gesture recognition for the camera with the target.

The control flow graph for the states of the distributed gesture recognition system is as illustrated in Figure 11(b). In each system state, the system forks to three different composite services, the intraframe service, interframe service, and display service. These services exist concurrently, and their data dependency is shown in Figure 11(c). The interframe service takes the result from intraframe service, and the display service consumes data from both intra- and interframe services. All the services in Figure 11 are composite services, and the aggregated components are different for different system states.

Suppose the two cameras exchange data information after the ellipse-fitting service, the intraframe service can be further decomposed into services, as shown in Figure 12. For the control flow graph (Figure 12(a)), the system first fork two concurrent services for both cameras, and join the two services after ellipse fitting. The fusion service comes after the join operation to integrate the data from both cameras. The data dependency shown in Figure 12(b) comes directly after the control flow.

6.5 Authority Control Protocol

An authority control protocol designed for the distributed gesture recognition system is presented in this section. We verified this protocol through a model-checking and verification tool: *SPIN* [Holzmann 2004].

Ubiquitous sensor systems require peer-to-peer computing, along with efficient but sophisticated communication protocols. These protocols find use in real-time systems, which tend to have stringent requirements for proper system

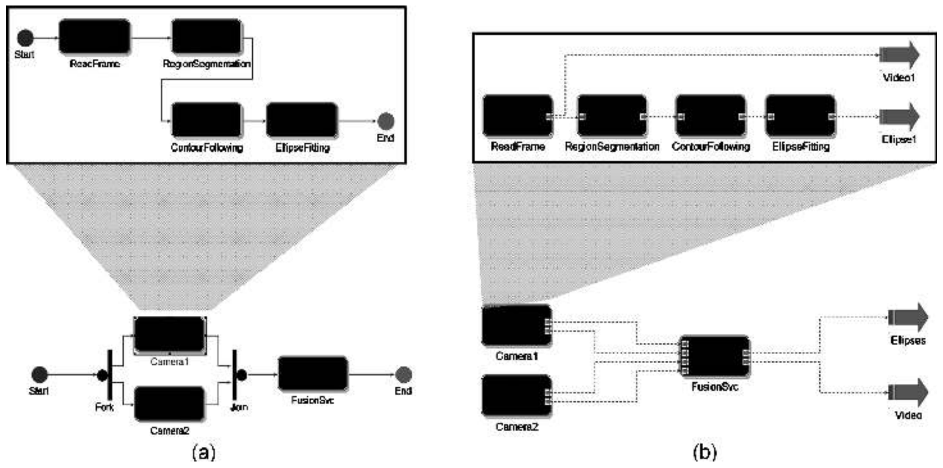


Fig. 12. The (a) control and (b) data flow service graph for the composite intraframe services in the recognized by both state.

functionality. Hence, the protocol designed for these systems not only necessitates transcending typical qualitative analysis using system simulation but also requires verification. The protocol must be checked and verified to ensure it does not cause unacceptable behaviors, such as deadlocks or process starvation, and has correctness properties, such as fairness. Formal verification of detailed system functionality is costly and time-consuming, if not impossible. A more realistic verification for communication protocols is to use abstract modeling. Each node participating in the protocol is modeled as a finite-state automaton, and the messages passing between node pairs are modeled using abstract channels with or without buffers. Verification tools are then used to exhaustively check the correctness of interactions among the finite automata.

When multiple cameras are used, camera nodes have to exchange data and control information about captured video streams with their neighbors. This communication helps perform recognition as a whole and eliminate redundant operations. The entry points of each processing stage are candidates for data exchange. Hence, we assume communication occurs after intraframe processing. In addition to recognized body parts, camera nodes also exchange control messages to determine which node shall perform gesture recognition for a certain person. The camera orientation is assumed to remain stable during the experiment. The control messages consist of timestamps for synchronization and control tokens to determine the ownership of detected persons. As a result, only the owner camera performs interframe processing. When a node receives packages from its neighboring nodes, the received data has to be combined with its own captured dataset and uses the received control signals to determine the successive procedures. The camera nodes first find the matching body parts within the received and captured datasets, and the one that has the most pixels in the *head* part is considered as the dominant camera. Camera nodes then

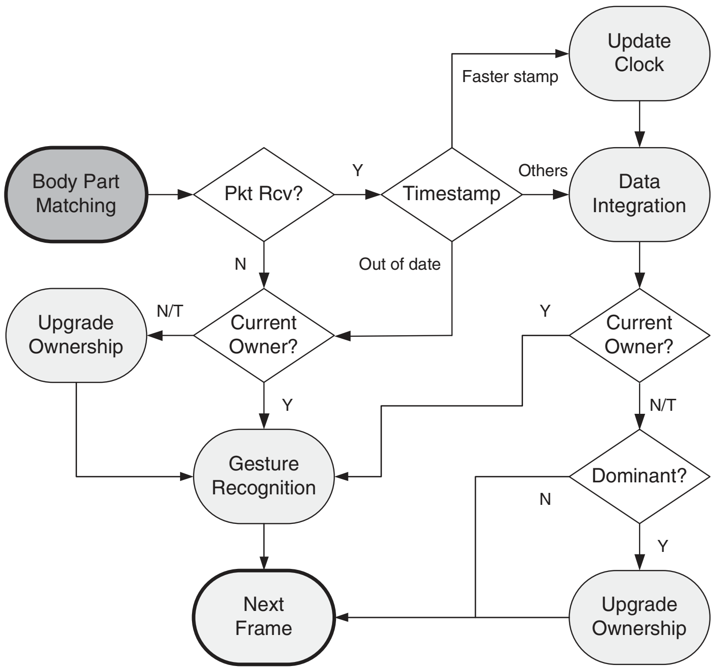
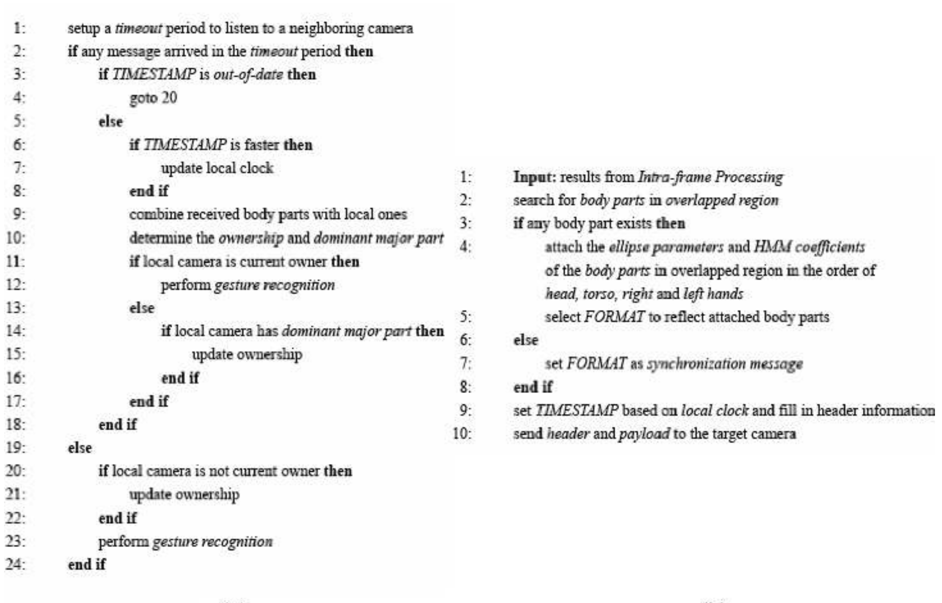


Fig. 13. The control authentication protocol for distributed gesture recognition.

use the received control messages and the dominant camera to determine the ownership of detected persons.

Although the gesture recognition system can handle multiple simultaneous objects, in the process of control authentication, each target obtains its ownership information independently of the awareness of other objects. As discussed previously, target-centric modeling is used to determine the ownership of each person inside the scene. For each detected person, a separated service is spawned to recognize the gesture of the person. The services take received messages to identify the current system state for each target and perform operations accordingly. The cameras with the targeted person is identified first, and the camera that performs the gesture recognition is then chosen based on the proposed protocol, as illustrated in Figure 13.

We assume that the control and data message exchanges occur after intraframe processing. After graph matching, the camera nodes wait and check if there are pending data packets from their neighbors. If no message is received, or the received message has an out-of-date timestamp, the camera node performs gesture recognition on the captured body parts. If the target is not owned by the current node, the node claims temporary ownership, in case of delayed or lost messages from its neighbors. When a node receives messages from its neighbors, it first checks the timestamp of the packet and updates its own clock if a faster timestamp within a threshold is received. The body parts in the received messages are matched to the captured dataset, and the dominant camera for each detected person is then determined. There is an owner token



```
1:  setup a *timeout* period to listen to a neighboring camera
2:  if any message arrived in the *timeout* period then
3:      if *TIMESTAMP* is *out-of-date* then
4:          goto 20
5:      else
6:          if *TIMESTAMP* is faster then
7:              update local clock
8:          end if
9:          combine received body parts with local ones
10:         determine the *ownership* and *dominant major part*
11:         if local camera is current owner then
12:             perform *gesture recognition*
13:         else
14:             if local camera has *dominant major part* then
15:                 update ownership
16:             end if
17:         end if
18:     end if
19: else
20:     if local camera is not current owner then
21:         update ownership
22:     end if
23:     perform *gesture recognition*
24: end if
```

(a)

```
1:  Input: results from *Intra-frame Processing*
2:  search for *body parts* in *overlapped region*
3:  if any body part exists then
4:      attach the *ellipse parameters* and *HMM coefficients*
        of the *body parts* in overlapped region in the order of
        *head, torso, right* and *left hands*
5:      select *FORMAT* to reflect attached body parts
6:  else
7:      set *FORMAT* as *synchronization message*
8:  end if
9:  set *TIMESTAMP* based on *local clock* and fill in header information
10: send *header* and *payload* to the target camera
```

(b)

Fig. 14. The pseudocode for the (a) authority control protocol and (b) message preparation.

for each target, and the camera that currently owns the token performs gesture recognition for the targeted person. The ownership changes when the current owner no longer dominates the target for a certain period of time. For a camera without ownership tokens, if the camera dominates a target, it upgrades its ownership, otherwise it does nothing. These operations are summarized as the pseudocode in Figure 14(a).

To ensure correctness and sufficiency of the proposed protocol, we used *SPIN* to verify the system. We claim that the protocol is fault-tolerant to message losses. As long as a person is detected, there will be at least one camera that recognizes the person's gesture, and only one camera will be performing gesture recognition when all the messages are received correctly within a frame. Our claim is proved by using *SPIN*'s exhaustive verification. Each camera node is modeled as a concurrent procedure, and each procedure would change its state based on the received messages and branch conditions listed in Figure 13. We then use *SPIN* to exhaustively simulate all the possible permutations of the camera states. The verification result also shows no deadlock, redundant state, or undesired loop in our system.

6.6 Distributed Gesture Recognition Results

The distributed gesture recognition system runs on a set of Windows machines and uses Webcams to capture video streams. Internet is chosen as the communication channel of the prototype system. The camera nodes are connected within a local area network (LAN) and use the user datagram protocol (UDP) to transmit data and control messages between sensor nodes. For each message

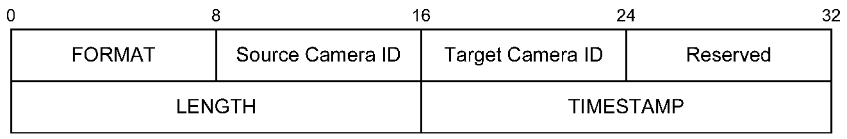


Fig. 15. The 8-byte header for data and control messages.

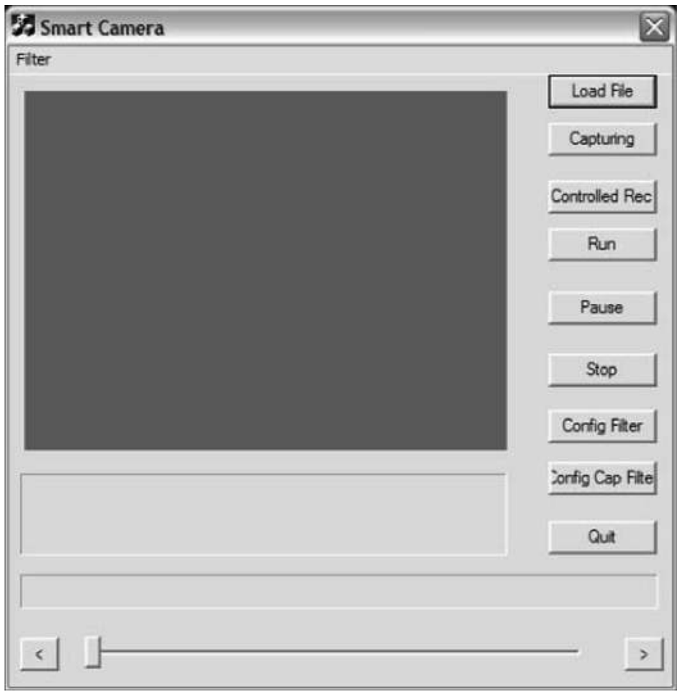


Fig. 16. A snapshot of the prototype program for our distributed gesture recognition system.

packet, it contains an 8-byte header, as shown in Figure 15, and a variable-sized payload. The header includes six fields, such as *FORMAT*, source and target *ID*s, *LENGTH*, *TIMESTAMP*, and reserved area. *FORMAT* defines the type of messages, along with the data format in the payload. Some format types are used as short commands with an empty payload, such as synchronization or acknowledgement, while others define different data types and formats in the payload area. *TIMESTAMP* field is used for synchronization and to determine if a message is out-of-date and can be ignored.

The prototype gesture recognition software is written in C++ using the Microsoft DirectX libraries as the video-developing environment. A snapshot of the prototype software is shown in Figure 16. The video-processing services are registered as a DirectShow filter, and DirectX application program interfaces (APIs) are used to decode the input video streams and render the processed video outputs.

Figure 17 displays snapshots from the distributed gesture recognition system. The two video streams are taken from cameras with slightly overlapped

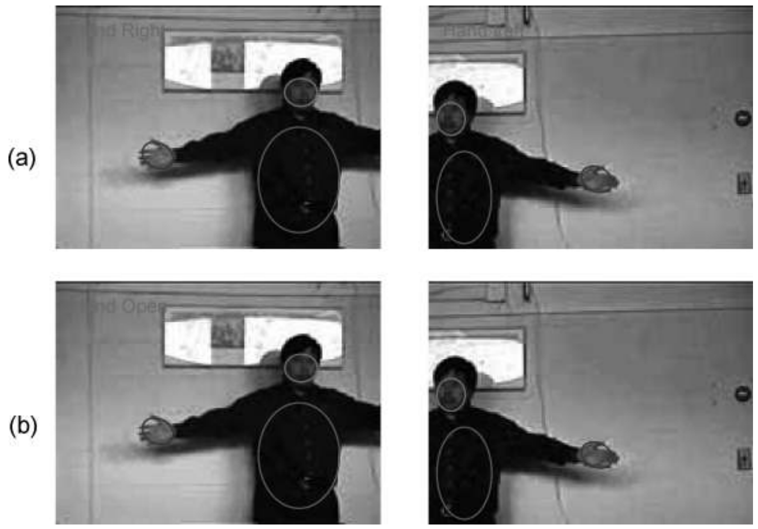


Fig. 17. The snapshots from the distributed gesture recognition system using (a) single-camera and (b) multiple-camera algorithms.

FOVs. If only the single-camera algorithm is used to recognize the movements of both streams, as shown in Figure 17(a), the left clip would be detected as *hand right*, while the right clip as *hand left*. However, the distributed system would take information from both streams and determine the entire movement as an *open hand*, as shown in Figure 17(b). The current version of the distributed recognition system can run at 15.23 frames per second on a Pentium III 1GHz PC with 128MB RAM.

According to the authority control protocol defined in Section 6.5, each pair of camera nodes with overlapped FOVs shall exchange data and control messages every frame after intraframe processing. For our prototype system, the cameras send out messages every 1/15 seconds, and the size of packets is content related. When there is no object moving around the overlapped region, only the synchronization message is needed: an 8-byte short command. Otherwise, when people appear in the overlapped region, the information of body parts are attached after the header. The payload includes maximum four sets of body part information: 64 bytes of eclipse parameters and 32 bytes of HMMs' coefficients for each body part. Figure 14(b) shows the pseudocode for preparing the data and control messages. The actual amount of data passed around the network ranges from 240 to 11,520 bytes/sec for each camera pair, depending on the content of each frame.

7. CONCLUSIONS AND FUTURE WORK

Large networks of physically distributed cameras cannot be effectively managed by a centralized server. Distributed algorithms allow us to build scalable systems of cameras that can analyze imagery in real time. We presented in this article a methodology to transform a well-defined single-node algorithm into distributed sensor framework. This methodology can help reduce the effort in designing a distributed sensor system when well-defined single-node application exists. A service-oriented software architecture is introduced for ubiquitous sensors. In such an architecture, a system can be modeled as the combination of the finite-state machine of the system, services, and middleware. According to the state of the system, the middleware dynamically binds different sets of services to perform required functionalities. We propose a distributed gesture recognition system as a design example. Our system use Windows machines and Webcams to capture video streams. Sensor nodes exchange data and control messages with neighbors to maintain broader view of the environment, and the authority control protocol used is also presented and verified using existing model-checking tools.

For future work, we would like to work on distributed-scheduling, power-management, and automatic calibration algorithms, as well as apply other applications to our ubiquitous framework such as tracking, face, and gait recognition. Ultimately, we would like to have more cameras in a wider area using wireless ad-hoc network to perform various in-network processing applications.

REFERENCES

Ballard, D. H. and Brown, C. M. 1982. *Computer Vision*. Prentice-Hall, Englewood Cliffs, NJ.

Bove, V. M. and Mallet, J. 2004. Collaborative knowledge building by smart sensors. *BT Tech. J. 22*, 4.

Bramberger, M., Doblander, A., Maier, A., Rinner, B., and Schwabach, H. 2006. Distributed embedded smart cameras for surveillance applications. *IEEE Comput. Mag. 39*, 2, 68–75.

Bramberger, M., Quaritsch, M., Winkler, T., Rinner, B., and Schwabach, H. 2005. Integrating multi-camera tracking into a dynamic task allocation system for smart cameras. In *Proceedings of the International Conference on Advanced Video and Signal-Based Surveillance*. IEEE, Los Alamitos, CA.

Caelli, T. and Bischof, W. F. 1997. *Machine Learning and Image Interpretation.* Plenum Press, New York.

Cai, Q. and Aggarwal, J. K. 1998. Automatic tracking of human motion in indoor scenes across multiple synchronized video streams. In *Proceedings of the International Conference on Computer Vision.* IEEE, Los Alamitos, CA, 356–362.

Cheong, E., Liebman, J., Liu, J., and Zhao, F. 2003. Tingals: A programming model for event driven embedded systems. In *Proceedings of the ACM Symposium on Applied Computing*. ACM, New York.

Collins, R. T., Lipton, A., and Kanade, T. 1999. A system for video surveillance and monitoring. In *Proceedings of the International Topical Meeting of Robotics and Remote Systems.* American Nuclear Society, La Grange Park, IL.

Collins, R. T., Lipton, A. J., Fujiyoshi, H., and Kanade, T. 2001. Algorithms for cooperative multisensor surveillance. *Proceedings of the IEEE 89*, 10, 1456–1477.

Davis, L. S., Borovikov, E., Cutler, R., and Horprasert, T. 1999. Multi-perspective analysis of human action. In *Proceedings of the International Workshop on Cooperative Distributed Vision*.

de Niz, D. and Rajkumar, R. 2003. Time weaver: A software-through-models framework for realtime systems. In *Proceedings of the Languages, Compilers and Tools for Embedded Systems*. ACM, New York.

Fleck, S. and Strasser, W. 2005. Adaptive probabilistic tracking embedded in a smart camera. In *Proceedings of the Conference Computer Vision and Pattern Recognition*. IEEE, Los Alamitos, CA, 134–141.

Grieffenhagen, M., Comaniciu, D., Neimann, H., and Ramesh, V. 2001. Design, analysis, and engineering of video monitoring systems: an approach and a case study. In *Proceedings of the IEEE*, 10, 1498–1517.

GUIBAS, L. J. 2002. Sensing, tracking, and reasoning with relations. *IEEE Signal Process. Mag. 19*, 2, 73–85.

HATCLIFF, J., DENG, X., DWYER, M. B., JUNG, G., AND RANGANATH, V. P. 2003. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the International Conference Software Engineering*. IEEE, Los Alamitos, CA.

HENGSTLER, S. AND AGHAJAN, H. 2007. Application-oriented design of smart camera networks. In *Proceedings of the International Conference Distributed Smart Cameras*. ACM, New York, 12–19.

HOLZMANN, G. J. 2004. *The Spin Model Checker - Primer and Reference Manual*. Addison Wesley, Boston, MA.

HOLZMANN, G. J. AND JOSHI, R. 2004. Model-driven software verification. In *Proceedings of the 11th Spin Workshop on Model Checking Software*. Springer, Berlin, 77–92.

KARSAI, G., SZTIPANOVITS, J., LEDECZI, A., AND BAPTY, T. 2003. Model-integrated development of embedded software. *Proceedings of the IEEE 91*, 1, 145–164.

KUSHWAHA, M., AMUNDSON, I., LIN, C. H., KOUTSOUKOS, X., NEEMA, S., SZTIPANOVITS, J., AND WOLF, W. 2006. An object-centric programming framework for ambient-aware, service-oriented sensor networks. In *Proceedings of the Information Processing in Sensor Networks*. ACM, New York.

LIN, C. H., LV, T., OZER, I. B., AND WOLF, W. 2004. A peer-to-peer architecture for distributed real-time gesture recognition. In *Proceedings of the International Conference Multi-Media and Exhibition*. IEEE, Los Alamitos, CA.

LIN, C. H. AND WOLF, W. 2005. A case study in clock synchronization for distributed camera systems. *Proceedings of SPIE 5683*. SPIE, Bellingham, WA.

LIN, C. H., WOLF, W., DIXON, A., KOUTSOUKOS, X., AND SZTIPANOVITS, J. 2006. Design and implementation of ubiquitous smart cameras. In *Proceedings of the International Conference Sensor Networks, Ubiquitous, and Trustworthy Computing*. IEEE, Los Alamitos, CA.

MATSUYAMA, T. AND UKITA, N. 2002. Real-time multi-target tracking by a cooperative distributed vision system. *Proceedings of the IEEE 90*, 7, 1136–1150.

MITTAL, A. AND DAVIS, L. 2001. Unified multi-camera detection and tracking using region matching. In *Proceedings of the Workshop on Multi-Object Tracking*. IEEE, Los Alamitos, CA, 3–10.

OZER, I. B., LV, T., AND WOLF, W. 2005. Design of a real-time gesture recognition system. *IEEE Signal Process. Mag. 22*, 3, 57–64.

OZER, I. B. AND WOLF, W. 2001. Video analysis for smart rooms. In *Proceedings of the Internet Multimedia Management Systems II*. SPIE.

OZER, I. B.,WOLF, W., AND AKANSU, A. N. 2000. Relational graph matching for human detection and posture recognition. In *Proceedings of the Internet Multimedia Management Systems*. SPIE, Boston, MA.

PENTLAND, A. 2000. Looking at people: Sensing for ubiquitous and wearable computing. *IEEE Trans. Pattern Anal. Mach. Intell. 22*, 1.

RINNER, B., JOVANOVIC, M., AND QUARITSCH, M. 2007. Embedded middleware on distributed smart cameras. In *Proceedings of the International Conference Acoustics, Speech, and Signal Processing.* IEEE, Los Alamitos, CA, 15–20.

STANKOVIC, J. A., ZHU, R., POORNALINGHM, R., LU, C., YU, Z., HUMPHREY, M., AND ELLIS, B. 2003. Vest: An aspect-based composition tool for real-time systems. In *Proceedings of the IEEE Real-time Applications Symposium*. IEEE, Los Alamitos, CA.

SVOBODA, T., MARTINEC, D., AND PAJDLA, T. 2005. A convenient multi-camera self-calibration for virtual environments. *Teleoperators Virtual Environ.* 14.

SZTIPANOVITS, J. AND KARSAI, G. 1997. Model-integrated computing. *IEEE Comput. Mag. 30*, 4, 110–112.

VELIPASALAR, S., LIN, C. H., SCHLESSMAN, J., AND WOLF, W. 2006. Design and verification of communication protocols for peer-to-peer multimedia systems. In *Proceedings of the International Conference Multimedia and Exhibition*. IEEE, Los Alamitos, CA.

VELIPASALAR, S., SCHLESSMAN, J., CHEN, C.-Y., WOLF, W., AND SINGH, J. P. 2006. Sccs: A scalable clustered camera system for multiple object tracking communicating via message passing interface. In *Proceedings of the International Conference Multi-Media and Exhibition*. IEEE, Los Alamitos, CA.

VELIPASALAR, S. AND WOLF, W. 2004. Recovering field of view lines by using projective invariants. In *Proceedings of the International Conference Image Processing.* IEEE, Los Alamitos, CA.

WATLINGTON, J. AND BOVE, V. M. 1997. A system for parallel media processing. *Parallel Comput. 23*, 12.

WILBURN, B., JOSHI, N., VAISH, V., LEVOY, M., AND HOROWITZ, M. 2004. High speed video using a dense camera array. In *Proceedings of the Conference Computer Vision and Pattern Recognition*. IEEE, Los Alamitos, CA.

WOLF, W., OZER, I. B., AND LV, T. 2002. Smart cameras as embedded systems. *IEEE Comput. Mag. 35*, 9, 48–53.

WREN, C. R., AZARBAYEJANI, A., DARRELL, T., AND PENTLAND, A. P. 1997. Pfinder: Real-time tracking of the human body. *IEEE Trans. Pattern Anal. Mach. Intell. 19*, 7.

Received May 2007; revised January 2008, August 2008; accepted March 2009