An Architecture for Semi-Automated Radar Image Exploitation

L. Keith Sisterson, John R. Delaney, Samuel J. Gravina, Paul R. Harmon, Margarita Hiett, and Daniel Wyschogrod

■ To improve the timeliness and accuracy of synthetic aperture radar image exploitation, the Defense Advance Research Projects Agency (DARPA) started the Monitor program at Lincoln Laboratory. This program was combined with related technologies from other contractors to develop the Semi-Automated IMINT (image intelligence) Processing (SAIP) system as part of an advanced-concept technology demonstration. The SAIP system accepts radar images in real time, distributes them to a number of algorithms for automated analysis, and organizes the images and algorithmic results for display to image analysts. The SAIP software infrastructure, which mediates between the operating system and the application code of the individual components, supports message passing between those components and allows the system to operate as a pipeline of parallel computation modules. In this article, we describe the design considerations of the overall system architecture. We also discuss how the component systems are organized and how the software allows many components from different organizations to work together.

The advent of low-cost, wide-area sensor platforms on unmanned air vehicles (UAV) will generate a profusion of data that dramatically increases the demands on image analysts. For example, Teledyne Ryan Aeronautical's Global Hawk UAV is designed to provide sustained high-altitude surveillance and reconnaissance at large standoff ranges. Its mission goals include the ability to loiter over a target area for 24 hours at an altitude of 65,000 feet. This UAV will carry a synthetic aperture radar (SAR) sensor that is projected to collect in one day enough data sampled at 1.0 m × 1.0 m resolution to cover 140,000 km^2 (roughly the size of North Korea).

To analyze the growing quantities of image data, the Defense Advanced Research Projects Agency (DARPA) initiated a project to develop the Semi-Automated IMINT (image intelligence) Processing, or SAIP, system. The SAIP system combines advanced automatic target recognition (ATR) algorithms and robust false-alarm mitigation techniques with commercial off-the-shelf computer hardware to filter out natural and cultural clutter, and to recognize and prioritize potential targets. It annotates image areas containing potential targets with target cues for the image analysts, who use visualization tools provided by the SAIP human-computer interface to establish true target types and produce a target report for the battlefield commander. The results of the exploitation process are saved and transmitted to other military systems.

In this article, we describe the design elements of the overall system architecture rather than the details of individual algorithms, unless their nature forces a particular architecture. We discuss how the component systems are organized and how the software allows many components from different organizations to work together. We also describe the parallel architectures of the detection and identification components—high-definition vector imaging (HDVI) and mean-square error (MSE) classification.

System Description

The SAIP system combines state-of-the-art automatic target-detection and recognition algorithms with human image analysts. The algorithms can examine large volumes of high-resolution imagery in great detail but have limited ability to put the imagery into context. Human analysts can use contextual cues and an overall sense of purpose to derive an accurate military analysis under difficult circumstances, but they cannot examine large volumes of high-resolution imagery in near real time for extended periods.

The SAIP system was conceived to accommodate the X-band radar planned for the Global Hawk platform. The Global Hawk flies at about 175 m/sec and illuminates a swath 10 km wide from an altitude of 20 km, resulting in an image pixel rate of 3,000,000 pixels/sec and a ground-area coverage rate of 2.3 km^2/sec [1]. These factors determined the SAIP requirements for computational power, data-transfer rates, and storage. The SAIP system must have a throughput capability that matches the imaging rate and it must also have a low processing latency. Systems analyses [2] of the requirements for surveillance systems of transporter-erector launchers (TELs, particularly those we know as SCUDs) found that the latency from the initial imaging of a target to the dispatching of a report from an image analyst needed to be less than five minutes. The latency of existing systems ranges from five to fifteen minutes [3, 4]. The under-five-minute constraint provided a latency goal for the SAIP system of two minutes for processing and two minutes for image-analyst exploitation. The imaging rate of the sensor makes an exploitable image about every minute, so with two analysts, each has an average of two minutes to exploit a scene while matching the latency requirements.

Figure 1 shows the assignment of the SAIP system components to the servers, and the data flow for the complete baseline system. It is composed of several symmetric multiprocessor servers and workstations interconnected by an asynchronous transfer method (ATM) network [5]. The SAIP software infrastructure supports message passing between the application code of the individual components supplied by different vendors and allows the system to operate as a pipeline of parallel computation modules.

The image receiver distributes complex-valued radar image data to registration and detection algorithms and up to three analyst workstations. The detector finds bright objects in the scene and computes characteristic features for each object. The locations on the ground of these detected objects are computed by using a sensor model calculated from the registration information. The list of detected objects, their features, and their geolocations are processed by algorithms aimed at false-alarm mitigation and change indication. These algorithms process only the detected objects, not all the imagery. The discrimination thresholding algorithm computes a score for each object by using the results from the preceding algorithms and rejects objects with a score below a threshold. The discrimination thresholding algorithm tries to lower the false-alarm rate by up to a factor of about one hundred. It also assigns a priority ranking for each detected object, or candidate target, that passes the threshold.

After the discrimination-thresholding algorithm, the HDVI module processes candidate targets in order of priority by using a complex image chip containing the detected object, followed by the application of an MSE classifier, which attempts to identify the object by comparing it to a stored library of signatures. A component called the MSE manager manages the work flow of the MSE classifier. Finally, the force-structure-analysis component analyzes the data to establish deployment patterns by comparing the distributions of detections to known deployment patterns of military forces such as tank battalions.

The results and the imagery from which they were derived are stored by the exploitable-image former component in a database for examination by the image analysts. This arrangement relieves the analyst from the necessity of keeping up in real time on a short-term basis and allows the analyst the flexibility to spend as long as necessary to exploit a scene.

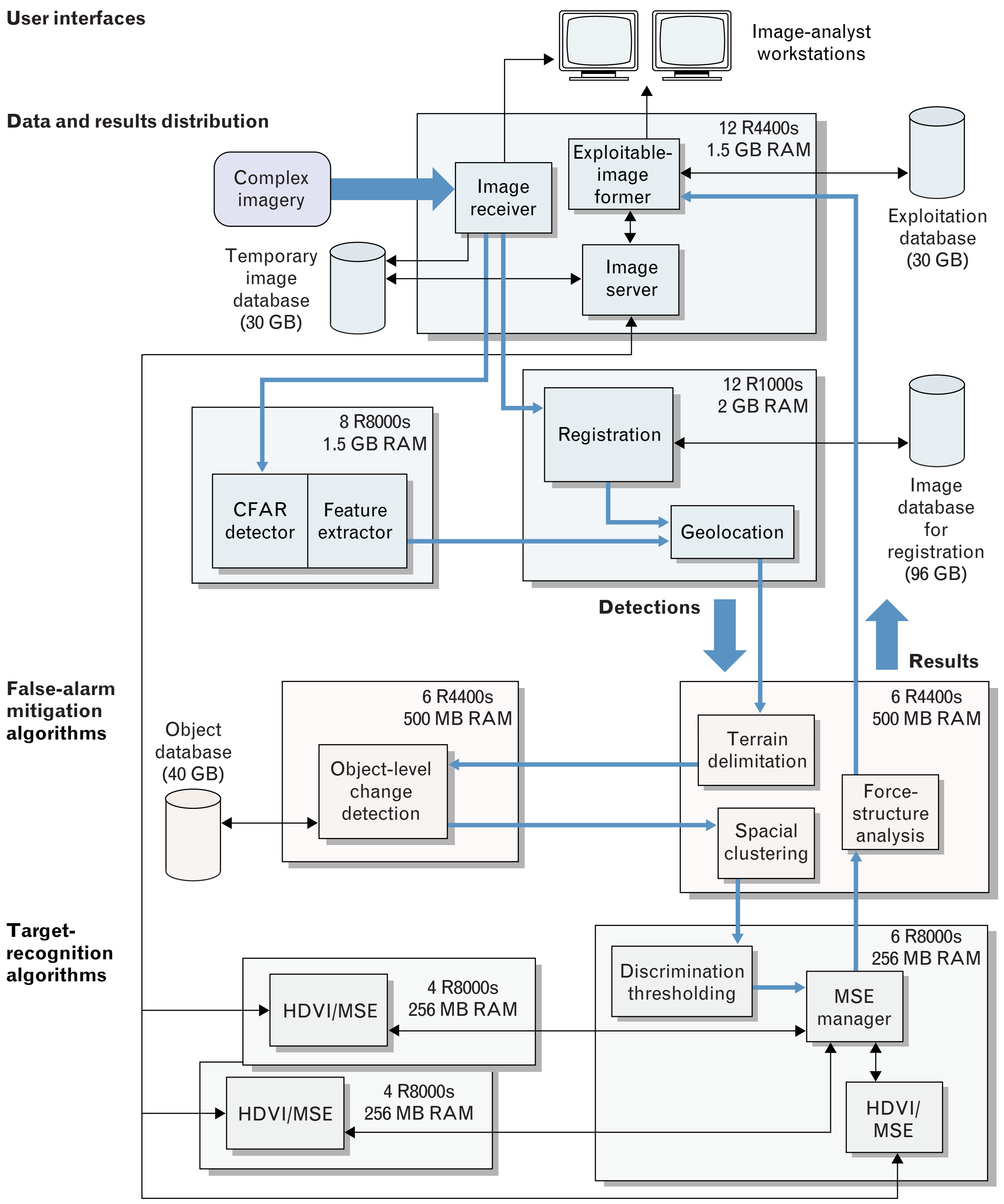


FIGURE 1. Semi-Automated IMINT (image intelligence) Processing (SAIP) baseline system to analyze image data and detect targets. Image data are processed to find bright objects and to determine the positions of these objects on the ground. The main data path through the system, highlighted in blue, starts with the complex imagery being sent to the image receiver. Each processing stage enclosed by a box represents a Silicon Graphics, Inc. server with the configuration indicated. Bright detected objects and their features are analyzed by false-alarm mitigation algorithms that reduce false alarms, recognize specific target types, and determine military force structures. In the target-recognition algorithms, the discrimination threshold ranks candidate targets before they are processed by the high-definition vector imaging (HDVI) and mean-square error (MSE) classifier. The results and corresponding original imagery are stored in the exploitable-image former to be retrieved by the image analysts.

Table 1. Parameters for Estimating SAIP Computational Complexity

Parameter	*Value*
Input pixel rate	3,000,000 pixels/sec
Real target density	25/km^2
Detection false-alarm rate	110/km^2
Area coverage rate	2.3 km^2/sec
Target-image chip size	31 pixels
HDVI pixel size reduction factor	3
Average size of region of interest (ROI)	4,000,000 pixels
Detected-object message size	500 bytes
Detections processed by HDVI/MSE	20%
Overview-display downsample factor	12
Image-frame interval	3

The computational complexity of each algorithm and the data-transfer rate required between algorithms determine the nature and organization of the hardware. Table 1 shows the design parameters used to estimate the computational complexity of Lincoln Laboratory SAIP algorithms, which are listed in Table 2. The main driving factor for the computational complexity differs for each algorithm. The image receiver and detection components process all the imagery, and their computational complexity depends primarily on the incoming pixel rate. After the detection stage, the remaining algorithms process messages containing feature information about the detections. The computational complexity of these algorithms depends on the detection rate.

The detection-rate model used for Table 2 assumes a dense target environment in a high-false-alarm region at a detection probability of 90%. The false-alarm rate represents a worst case typified by forested regions around well-populated areas, and was derived from work with data collected by the Advanced Detection Technology sensor developed at Lincoln Laboratory [6]. Figure 2 shows how the false-alarm density varies widely with location. Several algorithms need more than one processor (given the processor capability in 1997) to achieve the computation rates for handling the incoming image stream.

The image-receiver and detector components each process all imagery in various formats. Therefore, the communications data-transfer rate between these components is the highest in the system, and an 80-MB/sec high-performance parallel interface connects the servers used by these components. However, after the detection stage, the data-transfer rate between algorithms is moderate. While the complete system could be run within a single large processing system, the comparatively low interconnect bandwidth argues for the much less expensive solution of using several multiprocessor servers with a suitable network between them. This choice of architecture is reinforced by the practical need to have several independent organizations supply software components, some of which already exist for other environments, that will work together and have clean interfaces for straightforward integration. Thus the resulting architecture is a pipeline of parallel subsystems; each subsystem has enough parallelism to keep up with the average processing rate. The main connection between the servers is a 155-MB/sec ATM network.

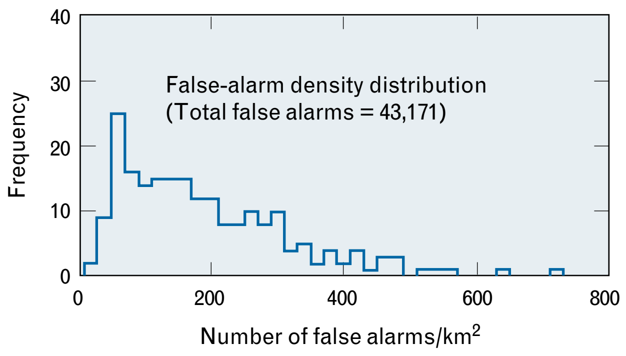


FIGURE 2. Area distribution of false alarms. The false-alarm density fluctuates widely as data are processed by the system. The average false-alarm density from the detector is 170 false alarms/km^2. The distribution has a much larger tail than an equivalent Poisson distribution. The variation in false-alarm density leads to a large variation in the processing load for those algorithms which depend upon the detection rate.

Table 2. SAIP Computational Complexity and Data-Transfer Rate Estimates

Processing Component	*Estimated Computational Loading (MFLOPS)*	*Driving Factor*	*Required Processors (Estimate)*	*Detection Output Rate (detections/sec)*	*Message Input Rate (MB/sec)*	*Message Output Rate (MB/sec)*
Image receiver	96	Pixel rate	8	NA	24	15
Image server	15	Pixel rate and detection rate	1	NA	0.003	8
Detector	150	Pixel rate	4	311	12	3
Feature extractor	124	Detection rate	3	311	3	0.2
Discrimination thresholding	31	Detection rate	1	54	0.2	0.03
HDVI/MSE classifier	1408	Real-target density and false-alarm rate from discrimination thresholding	20	54	2	2
Exploitable-image former	8	Pixel rate and force-unit density	1	1	6	0

From the beginning, we realized that a message-passing environment configurable for any combination of single-processor and multiprocessor nodes would provide flexibility in architecture and performance and allow independent development of the various components. Whether processes are executed on the same node or across a network should be a run-time configuration issue that does not affect the application code at all. This idea led to the development of the message-passing infrastructure components described in the next section.

Parallelism in Lincoln Laboratory components is achieved by many techniques. The choice of technique depends largely on the transfer rate of the incoming data. Table 2 shows the data-transfer rate estimates for the various processing stages of the SAIP system. When the incoming transfer rate is high, as for the detector stage, we achieve parallelism by logically dividing the image into range subswaths and performing the computations for each subswath on a different processor. Each processor shares the memory in which the image and results are stored to minimize data exchange between the activities on each processor.

In contrast, the HDVI/MSE classifier, with low bandwidth and high computation per object, achieves parallelism by distributing the computation for each object processed to a separate processor, which can be in a separate server. The amount of time SAIP spends transferring data to and from the HDVI/MSE module is small compared to the time it spends on computing. Shared memory is used for the reference image data when there is more than one processor in a server and the object data are passed by using the message-passing infrastructure. By allowing arbitrarily large numbers of servers and processors per server, this object-level parallelism is more efficient, more flexible, and much simpler to organize than a fine-grained parallelism in which individual computations such as scalar products are distributed across multiple processors.

While standards played little formal role in the development of the SAIP system, we considered the following guidelines to be important in controlling the complexity and volume of our work:

1. Although ANSI C++ was our preferred standard, we allowed ANSI C for some algorithm code, especially legacy code.
2. For data structures and methods, we defined a centralized library to provide common facilities to component developers, thereby providing a *de facto* coding standard. This library was built on the Standard Template Library (STL) [7, 8], which encouraged component developers to use the STL also.
3. We layered construction to hide communications methods from the application code.
4. We used commercial object-oriented database software for storing data.
5. We used commercial software for configuration control.

Although early in the development phase a decision was made to use Silicon Graphics, Inc. (SGI) workstations and servers, the Lincoln Laboratory software was written to be ported to alternative UNIX platforms at modest cost because only generic UNIX operating system functions are used and the code is ANSI C++ or ANSI C.

The remainder of this article explains the design and implementation of most of the Lincoln Laboratory components shown in Figure 1. We start with the infrastructure software that allows the components to communicate.

Infrastructure Software

The SAIP infrastructure components are the layered software common to most SAIP software components. The two major parts of the infrastructure are the SAIP interprocess communication (IPC) library and the SAIP data structures C++ class library, which both facilitate communication. A pair of UNIX processes can communicate by writing to and reading from a shared database, or by explicitly passing messages via the SAIP-IPC module.

Such interprocess communication can be defined and controlled on a case-by-case basis for each pair of communicating processes. However, the SAIP system standardizes interprocess communications by defining all data objects that can be stored in a database or passed in a message in a single C++ class library. In this library, the SAIP system also defines standard member functions for all objects such as storage allocation, message buffer packing and unpacking, and printing.

The single C++ class library approach has several advantages. First, standardization allows us to revise or reuse old code quickly when process implementations are changed or when new processes are added to the system. Second, experience gained from implementing the communication between one pair of processes readily transfers to the task of implementing the communications between two other processes. Third, standard methods for data classes speed the development of application code. Fourth, configuration control of class definitions is simplified by centralizing the definitions.

For the purposes of message passing, the SAIP C++ class library is layered on a C library for transferring data buffers between processes. The SAIP C++ class library hides much of the functionality of the IPC library to present application programmers with a simple communications model. In that model, C++ objects are passed between processes. An object is sent with one function call taking the object's address, a symbolic message type name, and a symbolic destination process name as arguments.

The SAIP-IPC Component

Each SAIP component application is composed of one or more application threads and one communication thread, as illustrated in Figure 3. See the sidebar entitled "System Terms and SAIP-IPC Design" for definitions of some computer-science terms needed to understand SAIP and a summary of SAIP-IPC design considerations. The communication thread establishes the socket connections to the other SAIP components and maintains the receiver data queues. All messages destined for a particular component are received and queued by the communication thread, which uses a separate queue for each source process. All messages from a component are sent directly by the component application thread. The separate receive communication thread implies the sends are nonblocking unless system buffers are exceeded. The application thread receives messages that have been queued up by the receive thread. Thus receive opera-

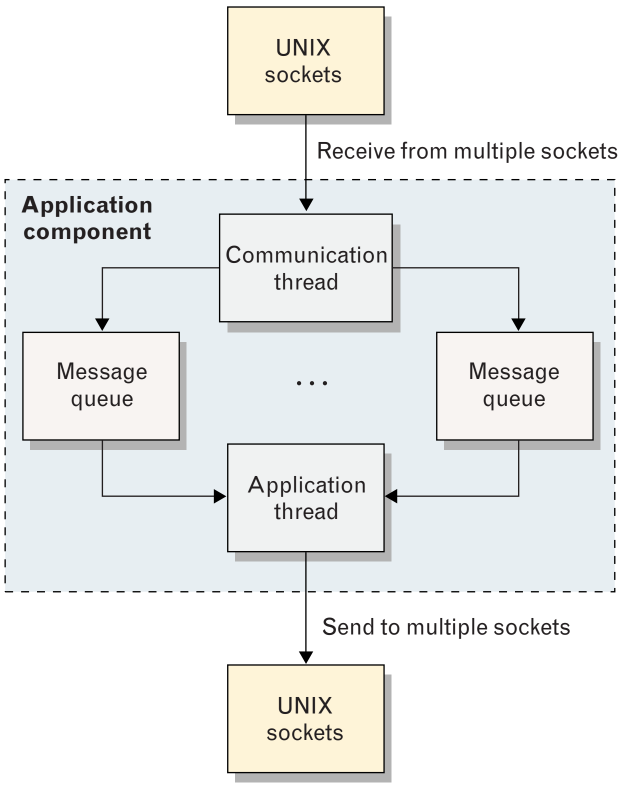


FIGURE 3. SAIP interprocess communication (IPC) structure showing the multithreaded nature of the component communication in SAIP. The communications thread forms a message queue for all other components that send a message to it. The communication thread also establishes the socket connections to the other SAIP components and maintains the receiver data queues.

tions are blocking only if there are no new incoming messages on the queue to be processed.

The communication thread forms a data channel, maintaining application-visible status and message-queue information on the health of the connections to other applications. Because queueing occurs at the receiver, each application can monitor the growth of its input queue and, if needed, take actions to process data more rapidly, typically by incomplete processing of each message until the backup is under control. This arrangement allows the application to receive data continuously until resource limits are exceeded. The sender will therefore block only when the receiver has stopped receiving data and the operating system buffers are filled. Thus the SAIP-IPC effectively supports nonblocking sends.

The communication thread uses sockets to implement the underlying communication mechanism. We chose sockets because they provide a generally uniform communication interface across almost all UNIX platforms. This design allows the application thread to select which inputs it wishes to deal with at any one time without blocking. In addition, it allows incoming data to be received while the application thread is processing previous message inputs.

The SAIP-IPC module is also responsible for launching the application components throughout the system, which is accomplished by a program launcher utility. Once launched, each application component invokes the initialization phase for the SAIP-IPC. The communication threads for the applications are started and the interconnections between the applications are established. The initialization algorithm is distributed across all of the components in the system that concurrently attempt to establish the socket connections defined by the applications. The SAIP-IPC module forms the connections independently of the detailed execution sequence of the set of applications forming the system. As a result the system launches quickly.

The Data-Structures C++ Class Library

The SAIP C++ class library uses the SAIP-IPC library to enqueue objects in each process as they arrive from other processes. Such enqueuing can occur in parallel with other computations because the communications thread handles it.

The library's *Message* class pulls together the functions layered on the SAIP-IPC library plus the global variables needed to coordinate them. There are methods to send and receive as well as to enter and exit a dispatching loop and the dispatch function. Objects are dequeued by the application thread by using one of two methods of the *Message* class. In the first method, a conventional receive function takes as arguments a symbolic message type name, a symbolic source process name, and a variable for storing the address of that type object arriving from that process. If there are no objects from that process enqueued when the function is called, it blocks until one arrives. In the second method, a function causes the thread in which it is called to enter a dispatching loop and to remain in the loop until explicitly exited. If there are

SYSTEM TERMS AND SAIP-IPC DESIGN

WE FIRST DEFINE some computer-science terms used with SAIP and then we summarize our design considerations for the SAIP interprocess communication (IPC).

Thread—A single sequence of execution within an application. All SAIP components have at least two threads, one for the main application and one that SAIP-IPC creates for receiving data (the communication thread).

Blocking—A thread will block, that is, stop executing, when it requires a resource that is unavailable. The typical resource that causes a thread to block is either data to be read that are unavailable, or data to be output when the output method is unavailable.

Socket—A UNIX mechanism for point-to-point data communication between two applications. If the applications are on the same computer, local sockets can be used. Generally, the applications are on different computers connected by a network that requires communication protocols to control the data flow.

SAIP-IPC Design

We had two design strategies for SAIP-IPC—use the emerging standard message-passing interface (MPI) [1] or use UNIX sockets within our software. At the start of this project it was important to establish a working IPC system so that applications could be connected to one another as soon as possible.

In 1995, we built a precursor system to SAIP called Monitor with MPI that had been developed by the University of Mississippi [2], and supported by Argonne National Laboratory [3]. In building Monitor, we uncovered several characteristics of MPI that made it unsuitable for developing a real-time system like SAIP.

First, although the implementation was based on UNIX sockets, only network-based sockets were used instead of local sockets between processes within the same multiprocessor system. This arrangement meant that all packets sent and received encountered all the additional overhead of network sockets, which is time consuming, and limited the speed of message transmission through the system.

Second, the MPI application interface assumed that all processes connect to all other processes in the system. In a network-based system this assumption increases system start-up time. Third, the implementation of MPI at the time was not thread-safe, so that sending and receiving messages would delay the application. Fourth, MPI queued messages at the sender rather than the receiver, so that a component could not know how much work was in its input queue.

Given these factors, we ultimately felt that MPI was not mature enough for our system. Instead, we decided to implement our socket-based approach to the SAIP-IPC module, paying close attention to the characteristics we needed.

References
1. Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *Int. J. Supercomputer Appl.* **8** (3/4), 1994.
2. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (MIT Press, Cambridge, Mass., 1994).
3. N. Doss, W. Gropp, E. Lusk, and A. Skjellum, "A Model Implementation of MPI," *Technical Report,* Argonne National Laboratory, 1993.

no objects enqueued when the function is called, the thread blocks until one arrives. If one or more objects are enqueued, the thread picks one fairly and then calls a dispatch function. When the dispatch function returns, it repeats the above procedure unless the dispatch function makes an explicit exit call.

The dispatch function's declaration is part of the SAIP C++ class library, but not its definition. For each process, a dispatch function must be supplied that is tailored to the purposes of the process. Typically, a dispatch function's top level is a switch statement on the symbolic message type name. The various case clauses in the switch statement handle objects of corresponding types. In this way, the processing of each object type becomes modular.

Most commonly, the first dequeuing function is called when one process is started to wait for initialization information from other processes. After all initialization is completed, the second dequeuing function is called and does not return until the system is shut down.

Pointer Semantics. Most objects handled by SAIP software components are composed of subobjects. For example, a detection object includes distinct subobjects holding the results of applying various algorithms such as spatial clustering or MSE target classification. The SAIP C++ class library uniformly uses pointer semantics to represent such inclusion. That is, an object contains pointers to its subobjects rather than explicitly containing those subobjects, and a highly composed object is the root of a tree of subobjects, sub-subobjects, and so on.

We found two advantages to designing the library with pointer semantics. First, pointer semantics permit different instances of the same object type to use different amounts of storage, either by pointing to different numbers of subobjects or by pointing to subobjects that use different amounts of storage. For example, detection objects passed between software components near the head of the SAIP algorithm chain typically use less storage than those passed between components near the end of the chain because the former contain null pointers, whereas the latter contain pointers to filled-in instances of results from later-applied algorithms such as the MSE classifier. Also, imagery objects can contain pointers to arrays of pixel attributes with contextually determined sizes.

Second, pointer semantics permit a clear indication of which object's subobjects have been set. If a pointer to a particular subobject is null (meaning empty, or zero), then the associated subobject's components are not set. When containment semantics are used instead of pointers, either subobjects must contain flags to indicate whether their components are set or reserved component values must be used as indicators that components are set.

As with most design commitments, there are disadvantages as well as advantages to handling subobject inclusion via pointer semantics. The most obvious disadvantage is the level or levels of indirection introduced in accessing those fundamental objects (e.g., integers, characters) which are the leaves of the object-subobject-… trees. It is not, however, the most significant disadvantage, as the following discussion indicates. The data buffers transferred between processes through the SAIP-IPC library are linear arrays of bytes. But a highly composed object is not a linear array; it has a number of characteristics for which the library must allow. First, the storage allocated for the object and its subobjects need not be contiguous; it could be scattered all over the local heap. Second, some storage holds pointers that are meaningful in the sending process but not in the receiving process. Third, the C++ compiler or run-time environment may insert hidden elements into its representation of an object (particularly pointers) that are not equivalent from one process to the next.

Thus the transferring process must flatten or serialize object-subobject-… trees into a linear form that includes all leaf object values plus enough structural information for the receiving process to unflatten the tree. The manipulations required for flattening and unflattening take computational time, particularly in copying leaf object values to or from a data buffer and in allocating storage for each subobject, subsubobject, and so on in the receiving process.

Those familiar with pointer semantics may have noticed something odd in the above discussion. It mentions only object-subobject-… trees, but pointer semantics allow for graphs of object-object pointers, even cyclic ones. By convention, the SAIP C++ class library restricts the use of pointer semantics in the library to constructing trees with leaf-ward pointers only. This restriction keeps the flattening and unflattening of objects relatively simple. For the information being communicated between SAIP software components, the restriction is not burdensome.

The C++ language allows us to associate functions with objects and to use the same function name and formal parameter list for a different function, depending on the associated object's type. The SAIP C++ class library uses this capability to provide uniform calls for functions common in general purpose to all object types but different in implementation details. The generic print function is a good example. Each object type defines a function with that name and with the same formal parameters. It is used to print the components of any object of that type and, because objects of different types have different components, the detailed implementations of the functions differ for different object types.

The C++ language does not allow us to similarly associate functions with fundamental types. It does, however, allow us to define function templates applicable to all types and to define specialized implementations (specializations) for particular types, including the fundamental types. The SAIP C++ class library uses this capability to provide uniform calls for functions that are common in general purpose to all types (not just all object types) but different in implementation detail. Again, the generic print function is a good example. The SAIP C++ class library defines a print-function template that provides uniform semantics for printing any data item. If the data item is an object, invoking the print-function template with the data item as an argument simply invokes the print-member function for that object's type. If the data item is a fundamental type, the appropriate specialization is invoked.

Such uniformity in commonly used function semantics allows us to write code that is readily comprehended and modified. Of course, the number of functions we would want applicable to all objects and fundamental types is not large. In the SAIP C++ library, generic functions are supplied for printing, heap and database storage allocation, logical and arithmetic operators, flattening data items into buffers and unflattening them out of buffers, and generating unique instances for testing.

Examples of Library Class Members

Most SAIP C++ class library members can be viewed as parts of several hierarchical classes used to compose the top-level objects passed between SAIP software components. While distinct at the higher levels of the hierarchy, they run together at lower levels. The highest levels of the hierarchies typically correspond to messages passed from one SAIP software component to another. This section describes some of these classes to illustrate principles of their construction.

Frame Class. The *Frame* class represents chunks of input-image data or collections of algorithmic results derived from the same chunk of input-image data. While a chunk of input data may correspond to an entire SAR spotlight-mode image or one whole patch of stripmap-mode imagery, nothing in the SAIP C++ class library requires such a correspondence. Data are framed for the convenience of the SAIP algorithms, although the framing is not completely arbitrary. The SAIP C++ class library does require that all the input image data associated with a frame be part of one SAR spotlight-mode image or one sequence of SAR stripmap-mode patches. Moreover, all input image data or derived results for only one SAR spotlight-mode image or one sequence of SAR stripmap-mode patches must be contained in sequential frames. Such a collection of frames is called a scene. The *Frame* class contains bookkeeping data for the frame, geometrical information about the location of the image and the radar, a pointer to the image array, and a pointer to a container class for the detections. This container class is an adjustable length vector of pointers to each detection. The ability to expand the container at will is convenient in the prescreening and feature-extraction software component in which detections are created and sorted geographically. The ability to shrink the container is convenient in the discrimination-thresholding component, which discards and sorts detections by discrimination score.

Detection. Detections are formed in the prescreening and feature-extraction software component, which defines each detection by an image location plus a number of characteristics that are used to initialize an instance of the *Detection* class. Later components in the algorithm chain translate the image location to the ground plane and add additional characteristics. In this way, the results of most SAIP algorithms are organized by detection. As mentioned above, detections are grouped by frame and carried down the algorithm chain by using *Frame* class instances as messages.

Exploitation-Related Classes. Two classes are used near the end of the algorithm chain to create data structures that help manage exploitation. The *Unit* class represents collections of detections presumed to be elements of a single military unit. Instances thereof are created by the force-structure-analysis software component. The region of interest (ROI) is represented by the *ROI* class, which consists of associated collections of detections with local imagery and other display-oriented information added. Instances of this *ROI* class are partially created by the force-structure-analysis component and are completed by the exploitable-image former.

Each *Unit* class instance indicates which detections compose the unit and which detections bind the unit (i.e., form the vertices of the unit's ground-plane convex hull). Units and ROIs are closely related. In fact, it is the presence of a unit that makes a region of imagery interesting. In that sense, each ROI contains a unit plus the imagery in which the unit was recognized. To support the efficient display of old and new imagery for an ROI, each ROI also indicates which old ROIs overlap significantly.

Algorithm Components

This section describes two of the Lincoln Laboratory SAIP components and focuses on how sufficient throughput is achieved, how data should be organized, and how the algorithm is organized for parallel computation.

Detection and Feature Extraction

The constant false-alarm rate (CFAR) detector consists of the CFAR algorithm and a feature extractor. It locates those portions of the input SAR images which are brighter than the surrounding background (i.e., they locate the man-made targets). Locating bright spots is accomplished by applying a CFAR stencil repeatedly over the image and computing the CFAR statistic for the pixel values, as described by the left-hand side of the equation

$$\frac{(x-\mu)}{\sigma} > \tau \,,$$

where x is the value of the sample pixel, μ is the mean of the pixel values, σ is the standard deviation of the pixel values in the stencil, and τ is the detection threshold. Figure 4 shows the CFAR stencil that is applied to locate bright spots.

Bright spots above threshold are clustered into blobs. Then a simple form of target-size discrimination eliminates blobs too small to be targets. The next step is to determine features from the remaining blobs. The resulting information is transmitted with *Detection* classes in a *Frame* message to the next SAIP component in the system. Clearly, clustering and feature extraction are dependent on the number of detectable objects in the scene; feature extraction on a frame of data can take longer than the CFAR algorithm and clustering.

The CFAR algorithm must keep up with the pixel rate of the input to the SAIP system. The approximate pixel rate needed in the baseline system is about one million pixels per second. In an unoptimized and unparallel form, the CFAR algorithm takes as much as one minute to process a frame of four million pixels on an SGI R8000 processor. After the optimization and parallelization described below, this algorithm takes about two seconds.

The largest improvement in CFAR processing speed came from optimizing the computation of the CFAR algorithm for the 64-bit shared-memory multiprocessor architecture of the SGI R8000 processor. Each of these processors has an on-chip primary data cache and a 4-MB off-chip secondary cache. The bulk

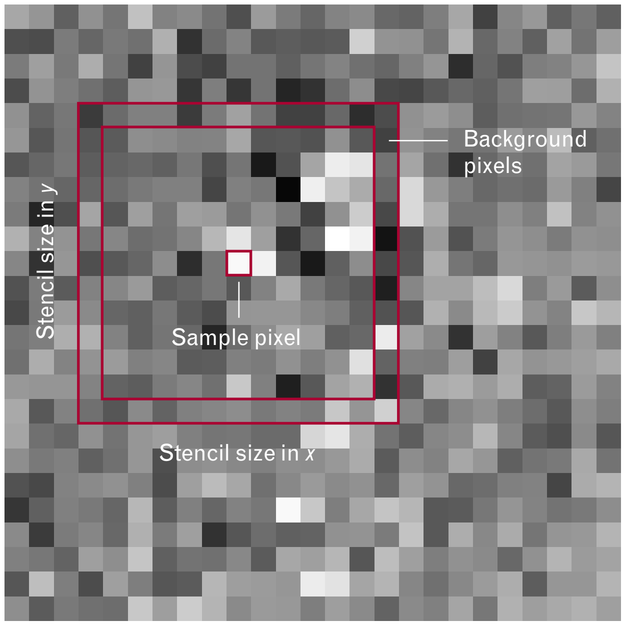


FIGURE 4. Constant false-alarm rate (CFAR) stencil applied to every pixel in the image to locate man-made targets. The targets are represented by those portions of the SAR image which are brighter than the surrounding background.

dynamic RAM (where most of the memory is located) is accessed off the processor boards via a 1-GB/sec backplane on four-way interleaved memory boards. Data must first be fetched from the bulk DRAM into the secondary cache, then read into the on-chip primary cache. Finally, the processor manipulates the data within its internal registers. All of these transfers are handled automatically by the cache control units. We access the memory space sequentially in an orderly fashion to optimize data flow within the memory subsystem, to allow the cache control units to prefetch data from DRAM in bursts, and to minimize cache flushing. In the context of handling images, we call this type of memory access rasterization.

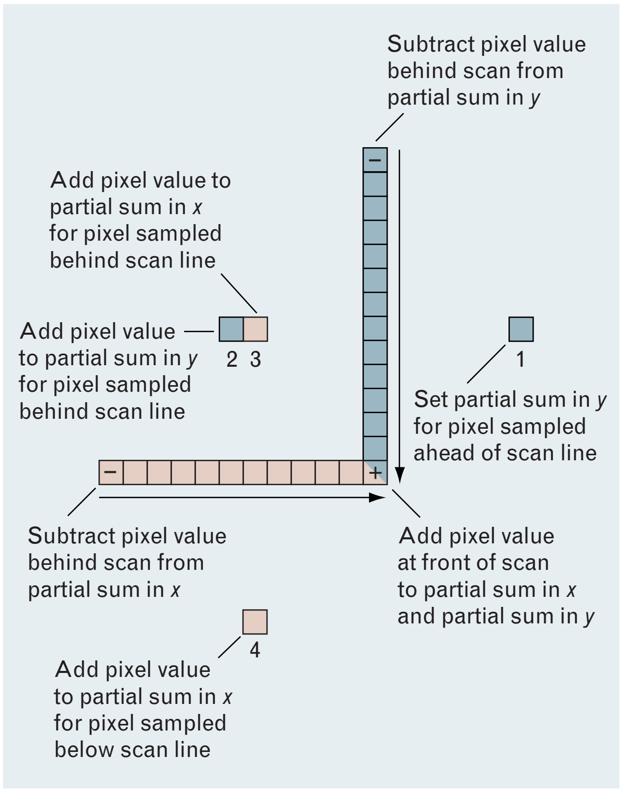

Subtract pixel value behind scan from partial sum in y
Add pixel value to partial sum in x for pixel sampled behind scan line
Add pixel value to partial sum in y for pixel sampled behind scan line
2 3
1
Set partial sum in y for pixel sampled ahead of scan line
–
+
Subtract pixel value behind scan from partial sum in x
Add pixel value at front of scan to partial sum in x and partial sum in y
4
Add pixel value to partial sum in x for pixel sampled below scan line

FIGURE 5. Scan pattern for the optimized CFAR stencil. The partial sums that form the mean μ and variance σ^2 are calculated for the original CFAR stencil legs in x and y. The order of the partial sums is important when forming the sums because the output buffers that store these sums are reused from previous frames of data. The output pixel labeled 1 is set first as the stencil is passed over the input image. The values in the pixels labeled 2, 3, and 4 are then added in sequence to the existing value in the output buffer to form the complete sum. To avoid counting the corner pixels twice, we make the x portion of the partial-sum scan two pixels shorter than the original CFAR stencil.

Thus to optimize CFAR, we rasterize the data accesses and keep the repeated memory accesses as close in proximity as possible. To do so, we must change the image scan pattern for the CFAR stencil. First note that the standard deviation σ is expressed by

$$\sigma^2 = E[x-\mu]^2 = E[x^2] - \mu^2 \ .$$

In the original relation

$$\frac{(x-\mu)}{\sigma} > \tau \ ,$$

we note that σ is costly to compute because a square-root operation is expensive in terms of cycle count. Therefore, we rewrite the relation as follows:

$$\frac{(x-\mu)^2}{\sigma^2} > \tau^2 \ .$$

This expression removes the square-root operation until the very end and can in fact be eliminated as long as the correct sign information is preserved (because there can be negative CFAR statistic values). It brings all the operations back to just multiplication, addition, and a single expensive division. When detecting bright pixels, we can avoid the division altogether. However, forming the CFAR image chips for the feature extractor requires the division. We can minimize the computation by not performing the division until the CFAR image chip is actually formed. The final relation is then given by

$$\frac{(x-\mu)^2}{(E[x^2]-\mu^2)} > \tau^2 \ .$$

The mean μ and the mean of the squares $E[x^2]$ can be calculated for the background pixels by taking advantage of the commutative property of addition and reordering the way the sums are computed over the stencil. Instead of summing around the stencil ring, we can calculate the sums in x and y independently (taking care not to sum the corner pixels more than once). The sum in x can be calculated by using a running sum and difference, as illustrated in the lighter-colored squares of Figure 5.

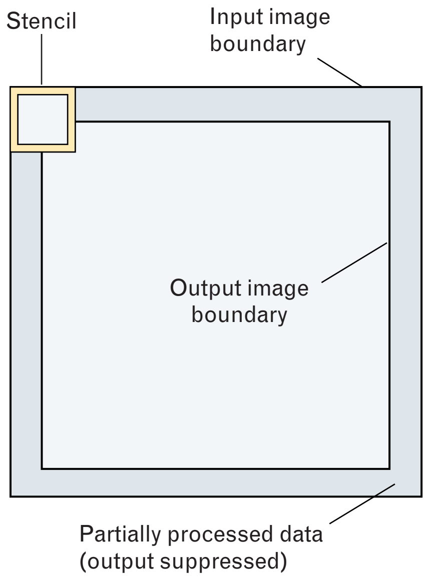


FIGURE 6. Relationship between input and output images associated with the CFAR stencil. The image that results from applying either the original or revised CFAR stencil is smaller than the original image. This reduction occurs because the stencil can examine only sample pixels that are at least half a stencil width away from the edge of the image.

A similar scan pattern can be applied for the partial sums in *y*; however, there is no advantage to scanning the *y*-direction the same way as the *x*-direction because that would not result in a continuous sequential access of the memory space. Instead, partial sums for the *y* legs of the stencil are maintained as in Figure 5.

Note that in this revised scan pattern, partial sums from each leg of the CFAR stencil are generated in the numbered sequence from 1 to 4 in Figure 5. When we deal with a single processor on a single image this order of sums is relatively unimportant. When we deal with adjacent search frames and a multiple processor application of the CFAR algorithm this order becomes important.

In addition to changing the CFAR scan pattern for the revised CFAR stencil, we can operate in parallel on a number of computers to optimize CFAR processing. The goals of parallel operation are to keep up with the overall data rate and to minimize the frame latency, which delays the entire system. We found that having multiple processors each process a portion of the incoming frame simultaneously is the best way to make the CFAR algorithm parallel. This approach is also used to cluster the bright spots.

The frames are divided along the range axis. The CFAR stencil is then run over each subimage by a separate processor and the results are merged. Figure 6 shows that running any stencil operator over an image leaves the output image pixels a subset of the input image pixels. Partial sums of pixel values at the range edges of the output image are suppressed (unwritten) so that an image subdivided among different processors does not interact at the boundaries between subimages.

Three processors handle this processing in the baseline implementation of the CFAR algorithm. Figure 7 shows that the input subimages are chosen such that the output images abut each other but do not overlap. Thus the separate processors can operate independently without the overhead of applying synchronization methods at the subimage boundaries.

We have less opportunity to optimize the clustering portion of the CFAR algorithm and the feature extractor. The clustering algorithm, which does not rasterize, can be thought of as a kernel-based operation on an image. Clustering assumes that a bright pixel has been detected at the center of the kernel and that other bright pixels within the kernel have already been detected and assigned to a cluster. The bright pixel at the center is assigned to one of the previously clustered pixels within the kernel. The feature extrac-

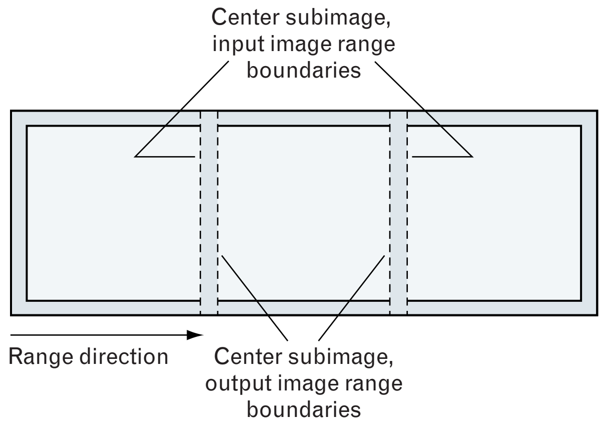

Center subimage, input image range boundaries
Range direction
Center subimage, output image range boundaries

FIGURE 7. Subimage processing. Each subimage is processed on a separate processor, and boundary regions must be merged correctly. Because only the input subimages overlap, the separate processors can operate independently without the overhead of applying synchronization methods at the subimage boundaries.

tor cuts out chips from both the original image and CFAR image and calculates the features of the clusters in them. Features calculated include mass (number of pixels), peak and mean CFAR values, and diameter.

We make clustering parallel in a manner similar to the CFAR algorithm. Unlike CFAR, however, this algorithm cannot be allowed to look across the range boundaries because pixels being clustered by one processor might affect the results in another. Thus the clustered output image is not complete for a range of half a kernel from the range boundaries on the subimages. The regions on the subimage range boundaries must be processed in a separate pass to complete the clustering for the frame.

We make the feature extractor parallel in a straightforward manner. Figure 8 shows the architecture for the CFAR algorithm and feature extractor. Cluster information from the CFAR and clustering stage are sent to a series of queues in a round-robin fashion, one queue per feature-extraction module. All the feature extractors work independently on their respective input queues. Special markers are inserted into each queue to indicate that there are no more clusters for a particular frame. When all feature extractors have processed all the inputs on their queue for a particular frame, CFAR recycles the data buffers. All the feature extractors output their results to their output queues. These output queues are then read in a round-robin fashion by a data collection module that constructs a *Frame* message with a detection vector and sends it to the next stage in the SAIP system.

Target Recognition

Target recognition is accomplished by using a pattern-matching algorithm. Detections that pass the discrimination thresholding module are resolution enhanced and compared to a library of SAR imagery of known vehicles. The vehicle that provides the best match is presented to the image analyst only if the match surpasses a quality criterion; otherwise the image is classified as unknown.

The target-recognition component makes extensive use of the STL container classes defined in the C++ standard. The dynamic memory handling and easy extensibility of the STL classes and C++ class features provide the flexibility needed to define a large library of vehicle imagery that can be retrieved on the basis of six independent variables for pattern matching in real time over a network of multiple processors.

Figure 9 shows the organization of the target-recognition component, which consists of the HDVI/MSE classifier and template library components. All detections associated with a given SAR image frame are collected in a *Frame* message and arranged in order of interest according to the criteria trained into the discrimination-thresholding algorithm. Only the

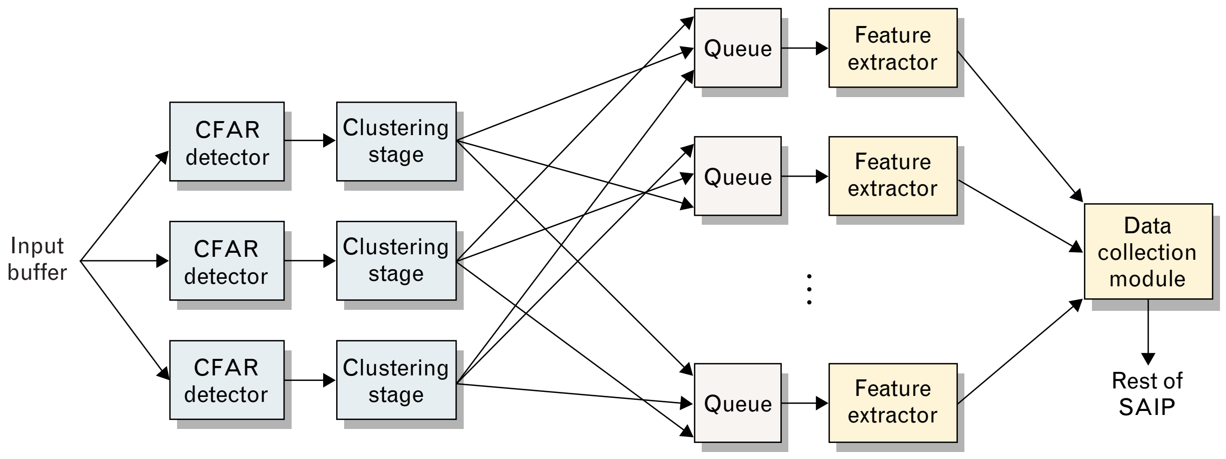


FIGURE 8. CFAR detector and feature-extractor parallel architecture. Information from the CFAR algorithm, including the clustering stage, is sent to queues in a round-robin fashion, one queue per feature extractor. All the feature extractors deliver their results to their output queues (not shown). These output queues are then read in a round-robin fashion by a data collection module, which forwards the results for processing through the rest of the SAIP chain.

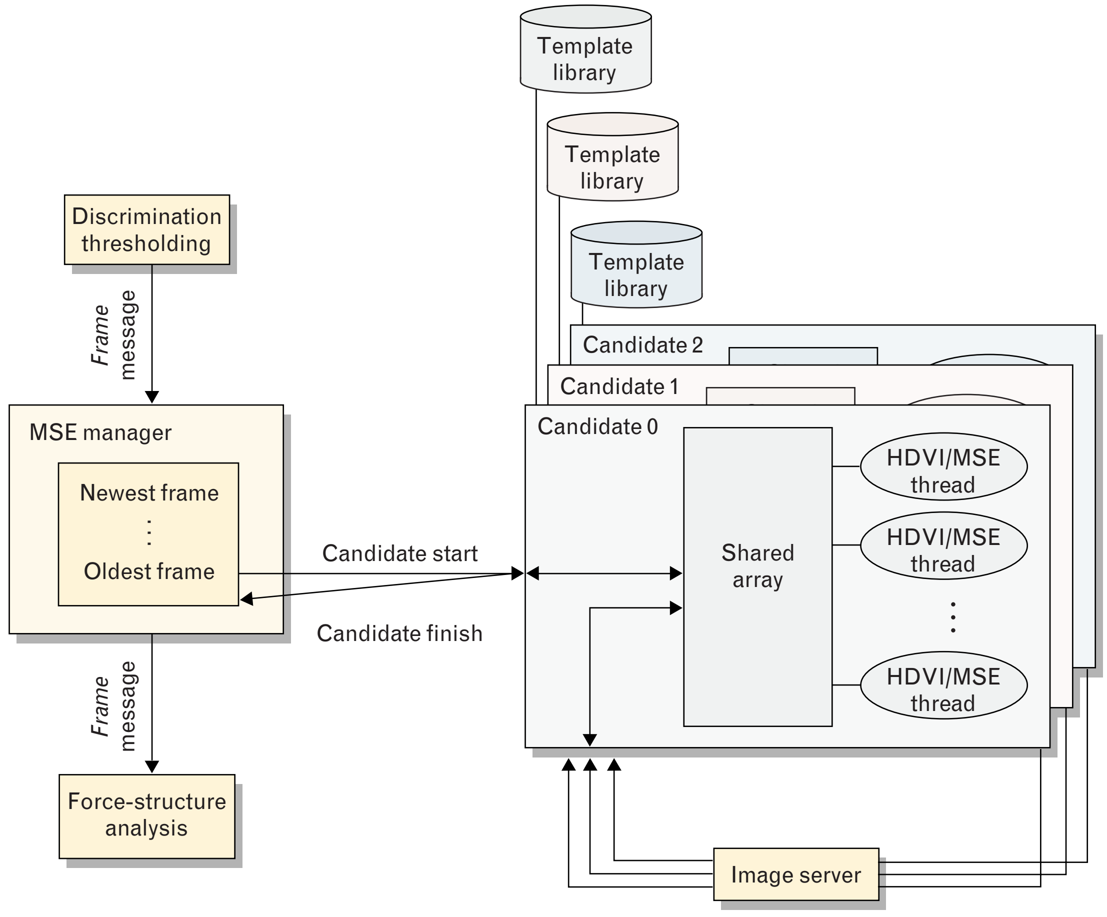


FIGURE 9. HDVI/MSE classifier algorithm consisting of HDVI/MSE threads and template library components. All detections associated with a given SAR image frame are collected in a *Frame* message and arranged in order of interest by the discrimination thresholding module. The MSE manager receives these messages and keeps track of processors, frames, and detections, and an arbitrary number of candidate processes. Each candidate process can contain an arbitrary number of threads and has its own copy of the template library. All detections, whether processed or not, are assembled into the *Frame* message and passed on to the force-structure-analysis module, which determines military force structures.

most interesting detections will be processed when time does not allow processing of all detections.

The results of the MSE classifier are placed in the detection class along with a resolution-enhanced image of the detection. All detections, whether processed or not, are assembled into the *Frame* message and passed on to the force-structure-analysis module.

The number of detections per frame that will pass discrimination thresholding and be processed within the time allotted to the classifier are not known until run time. Since it is also not known whether a particular detection will be processed, the *Frame* message is kept small by not storing the detection imagery within the frame. Thus an additional interface requirement of the classifier is that it must request detection imagery from an image server when necessary.

The fundamental task of the classifier algorithm is to find the vehicle template that, when compared with the detection, gives the lowest MSE [9]. The performance of this algorithm is improved by using the HDVI algorithm for resolution enhancement. In the SAIP baseline system the HDVI image of the detection and the templates are oversampled by a factor of three and compared to the original image. This arrangement results in nine times as many computations for a given template comparison, due to the increased number of pixels per image.

As a result, the HDVI/MSE classifier is two-stage multiresolution to reduce computation. In the first stage the classifier is run over the entire space of possible template matches at the unenhanced image resolution. The results of this classification are the starting point for classification with the HDVI enhanced imagery and templates. The second-stage search space is reduced to a volume around the answer found in the first stage, which significantly reduces the computation needed compared to searching the whole space with enhanced imagery.

The MSE classifier, which is designed to use any allocated hardware, works in parallel on any number of processors distributed on any number of networked servers. The various servers used do not even need to be running the same operating system.

Because the classification of each detection is independent of the other detections, each detection is assigned to an independent process. This level of algorithm segmentation results in efficient use of the processors: the number of detections available for processing will in general be larger than the number of processors available.

Two components in a dispatcher/worker model implement the classifier algorithm. The dispatcher module, called the MSE manager, handles the communications with other SAIP components shown in Figure 9, and a worker module candidate accepts work from the MSE manager and dispatches it a detection at a time to individual processing threads. Each candidate can run on a separate server with its own copy of the templates.

The MSE manager is a single threaded module that receives the frame of detections and distributes them to the candidate processes. As the processed detections are returned to the MSE manager, it places the results in the appropriate frame and sends the frames off to the force-structure-analysis module in the order in which they were received. The MSE manager keeps track of three main categories: processors available for classifying detections, frames from which detections are retrieved and stored, and detections that have been sent off for processing.

The MSE manager has no *a priori* knowledge of what process handles detections. When a process wants to work on detections it sends a *candidateDone* message to the MSE manager with no results in it. The MSE manager adds the detection process to its list. When the MSE manager receives a completed detection (a *candidateDone* message) the process that sent the message is available for processing another detection. The MSE manager then adds the process to its list of available processes. This technique makes it easy to add detection processes.

Because the candidate processes can be multithreaded, many instances of a candidate process can exist in the available processes list. To handle this dynamic data the STL class *multiset* stores the available processes. When a detection is sent off for processing, the process to which it is sent is removed from the available processes list. When the processed detection is received, its process becomes available again on the MSE manager list. This scheme allows a variety of processor configurations. If one of the detection processes should fail, then that process would simply not return its results, and all the other processes would be unaffected.

A *FrameAccount* class keeps track of each *Frame* message encountered to free the candidate processes of any order or time constraint, to minimize code complexity, and to allow for maximum flexibility in allocating CPU resources. The bookkeeping of the detections for a particular frame is done with the two STL *set* classes: *notSent* and *notDone*. When a frame is first encountered, an index for each detection to be processed is placed in both the *notDone* and *notSent* set. The *notSent* list is the place from which the MSE manager picks detections to send. When a detection is sent, its index is removed from this list. If a *candidateDone* message is received, its index is removed from the *notDone* list. If the *notDone* list is empty, then the frame is sent off to the force-structure-analysis module.

The MSE manager queues the incoming frames so that new frames can be received while older frames are being processed. The frames must be sent off in the order they are received for the benefit of other SAIP modules that depend on this order. No limit is placed on the number of frames that can be in the queue. The STL *list* class is used to keep track of the frames. The classifier algorithm is designed to process as many detections as possible within a fixed amount of time and to pass on the results even if all of the detections have not been processed. This feature is essential in a real-time system with a greatly varying processing load to prevent the backup of data during times of high detection density. Two parameters limit the number of detections that get processed. One parameter limits the number of detections to be placed into the *notSent* and *notDone* lists; another parameter specifies the maximum number of seconds a frame is held for processing.

Template Library

A key design issue for the MSE classifier was the use and storage of the templates. Individual templates must be easily retrieved by the software and the personnel maintaining the template library. In addition, the template library must grow and evolve as new templates are collected. Table 3 lists the six independent parameters that uniquely distinguish one template from another.

The template parameters used to distinguish the template—radar mode, squint angle, and depression angle—are compared to the respective values of the frame of detections being processed. Those templates which give the best match are used as the set to search. The parameter oversampling divides the templates into first-stage and second-stage templates. Two other parameters—class type and aspect angle—define the search space. The result of the search will be the class type and aspect angle that best matches the detection.

Because each template is described in a six-dimensional space, both disk and memory storage use a six-dimensional storage system. In both storage schemes each template is independently specified with all six dimensions.

The UNIX file system is used to create a six-dimensional database of template files. The templates are stored in a file path in which the parameter values are the directory names. Thus all of the templates for radar mode Globalhawk_spot with a squint angle of 7.3° are in the directory Globalhawk_spot/7.3/. Table 3 shows the order of the parameter directories. At the bottom of the directory tree is the actual template file whose name is the aspect angle of the template.

This system allows easy insertion and removal of the templates with standard UNIX file system tools.

Table 3. Template Parameterization Space

Parameter	*Number of expected values (determined by data, not software)*
Radar mode	9
Squint angle	3
Depression angle	3
Oversampling	2
Class type	30
Aspect angle	72

The classifier makes no assumptions about the scope of the templates; it determines which templates exist by reading the library. This determination allows for efficient disk usage and maximum flexibility in maintaining the library.

Table 3 also indicates that the total library planned for SAIP will have 349,920 templates ($9 \times 3 \times 3 \times 2 \times 30 \times 72$). At approximately 6 kB/template, the templates represent over 2 GB of data. This amount does not present a disk storage problem, but it can exceed the amount of memory dedicated to the classifier. Consequently, we adopted the strategy of reading in only templates that have a possibility of being used. Of the templates read, we group them in memory by aspect angle and class type so that all of the templates used in a particular search reside in the same region of memory. Thus even if the in-memory template library is larger then the physical memory available, the memory region in use for any particular frame will be small enough to avoid reading templates in from swap space.

Figure 10(a) shows that templates are stored as sparse matrices. The imagery is stored in array x; the coordinates of the pixel values are stored in arrays i and j. The sparse matrix storage is a fast way to do the comparison of the template with the detection image while excluding non-vehicle clutter pixels that do not contain any information about the vehicle. Otherwise, a clutter mask would need to be stored with the template and a test done on every template pixel before comparing that pixel to the detection image.

```
Template<class Type, class IndexType> class Sparse2Darray {
Public:
  Type *x ;         // values
  IndexType *i ;    // ith coordinate of values
  IndexType *j ;    // jth coordinate of values
  Unsigned int N ; // size of x, i, and j arrays
}
```

(a)

```
Class Plates6 : public IndexVector<RadarMode, Plates5> {} ;
Class Plates5 : public IndexVector<SquintAngle, Plates4> {} ;
Class Plates4 : public IndexVector<DepressionAngle, Plates3> {} ;
Class Plates3 : public IndexVector<Oversampling, Plates2> {} ;
Class Plates2 : public ExactIndexVector<ClassType, Plates1> {} ;
Class Plates1 : public IndexVector<AspectAngle, Plate> {} ;
Class Plate   : public Sparse2Darray<float, unsigned short> {} ;
```

(b)

FIGURE 10. C++ code for storing templates as sparse matrices according to (a) class type and (b) storing in-memory templates in a six-dimensional class that is inherited from the *Sparse2Darray* class.

Figure 10(b) shows that the in-memory templates are stored in a six-dimensional class that is inherited from the *Sparse2Darray* class. The template image intensities are stored as *floats* and the pixel coordinates as *shorts*. This format was found to give the best performance in the smallest memory space. The template class definition of *Sparse2Darray* makes it easy to change the data format should this ever be necessary. *Plates1* is a collection of *Plate* classes arranged by aspect angle. *Plates2* is a collection of *Plates1* classes arrange by class type or, equivalently, a two-dimensional collection of *Plate* classes arranged by class type and aspect angle. In this manner, the six-dimensional *Plates6* class defines the entire template library.

The *IndexVector* class, a special version of the STL vector class, is a vector of pairs. The first component of the pair specifies an index and the second component specifies the object associated with the index. The *IndexVector.operator(index)* function returns the object that most closely matches the index. Thus a call *Plates1(53.2)* returns the *Plate* class that has an aspect angle closest to 53.2°. The *ExactIndexVector* is the same but returns only an object that matches exactly. Thus the call *Plates2(bmp2)(53.2)* returns the *Plate* class that has an aspect angle closest to 53.2° from the *bmp2* class. If there is no *bmp2* class then it returns an empty *Plate*.

The *IndexVector* class is an enhanced version of the STL map class but is based on the STL vector because the STL vector guarantees that the items will be stored in contiguous memory. This is how templates for a particular search are kept together in memory.

The candidate component is a multithreaded process that receives detections for processing, distributes the detections to processing threads, and returns the processing results to the process that requested the processing. The candidate component does not know about the process that requests work. It knows only to send the results back to the process that requested them. This design leaves much flexibility to add both detection dispatchers and candidate processors as desired. Adding the site monitoring facility, which uses HDVI, is simply a matter of launching an additional candidate process that gets its messages only from the site monitoring dispatcher. No code changes are necessary. Also, the image analyst can request classification interactively by sending a message to a candidate process.

The candidate process is designed to be a per-server process. Each candidate process has a copy of the template library in memory to avoid reading templates over the network. The candidate component launches one thread per processor on its host server. The read-only templates are in shared memory and visible to all the threads. Figure 11 shows the candidate algorithm.

The *Work* class accomplishes the bookkeeping for the candidate component. Each thread launched has a version of the *Work* class. All the *Work* classes are collected together in an STL *vector* class, *WorkVector*, which resides in shared memory. Each thread examines *WorkVector*, selects a work area for its own use, and sets a flag so that no other thread will use this area. A semaphore protects this procedure so that only one thread is selecting a work area at a time. After this start-up procedure all of the threads have their own section of shared memory, which the other threads will not attempt to use.

The *candidateStart* message is received by the main thread, which retrieves the detection image from the image server. Since all the detection threads share the same network connection, there is no advantage to

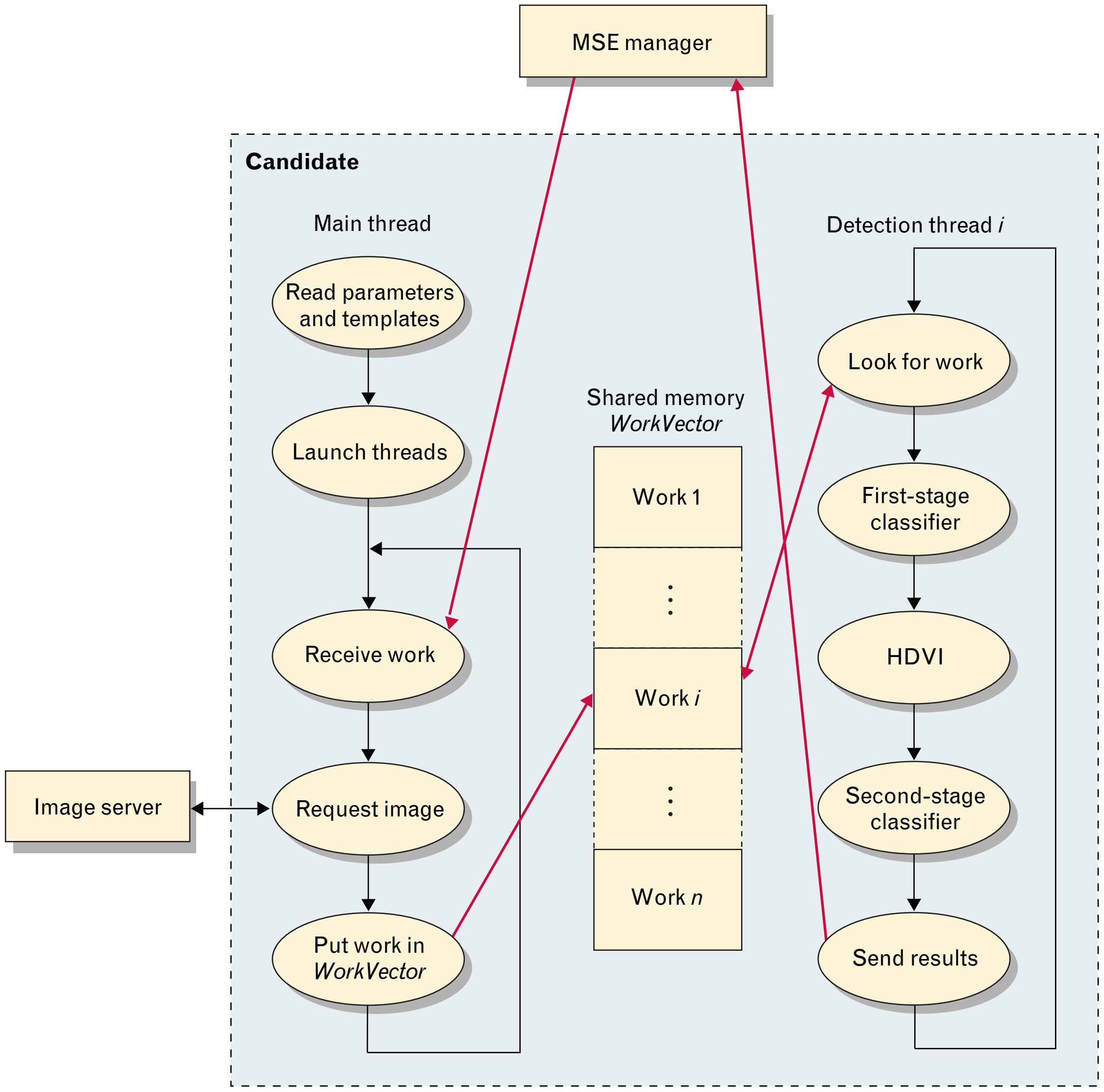


FIGURE 11. Candidate algorithm, a multithreaded process that receives detections for processing, distributes the detections to processing threads, and returns the processing results to the process that requested the processing. One detection thread is launched for each processor available on the server. The main thread handles communication with the image server and places work in the shared-memory area called *WorkVector*. Each detection thread owns a piece of the shared memory and completes the work placed there. When finished the detection thread returns the results to the process that originated the request.

having the individual detection threads request the imagery. The main thread decides which *Work* to put the *candidateStart* class in by looking for a *Work* that is idle.

The rest of the processing is up to the individual detection threads. They continually monitor their *Work* class until they find new work pending. The new work starts the cascade of processing that eventually fills out the fields in the *candidateDone* class. Once completed the thread sends the *candidateDone* message back to the process that originated the *candidateStart* message.

We assess the quality of the classifier dispatching algorithm by determining the processor efficiency: the time spent in calculation divided by the time needed per calculation. This parameter measures how well the available processors are utilized. Any unaccounted time is spent in message passing and system overhead of running multiple threads. Table 4 shows the execution times for each processing stage. Each candidate process has a 96% efficiency and the MSE manager has a 94% efficiency. The product of these two, 90%, is the overall efficiency.

Table 4. HDVI and MSE Processing Times

Processing Stage	*Execution Time (sec)*
HDVI	1.61
First-stage classifier	0.34
Second-stage classifier	0.13
Total for candidate process	2.08
Total time spent in thread	2.16
Amount of time per detection processor, measured by the MSE manager	2.3

Exploitation Management

At this point in processing, all algorithmic results have been calculated and all corresponding imagery is available at several resolutions in an image database. The results from these two sources must be merged for presentation to several image analysts. The software that performs this merging and distribution function is called the exploitation manager.

Exploitation-Manager Requirements

At one time, we considered organizing the software components of the SAIP human-computer interface around the ROI concept. Each ROI, a package of contiguous imagery and related algorithm information, would be presented as a separate exploitation task to an image analyst. The ROI concept, however, is not without drawbacks. For example, a general problem is how to handle imagery that is not part of any ROI. Should it be entirely unavailable to image analysts? Should it be accessible only by exploring outward from an ROI? Should it be available only on demand? Should image analysts be expected to exploit non-ROI imagery at low priority? Or should all contiguous imagery be presented to an image analyst as it arrives, with the ROI simply serving as cue?

We eventually decided that the SAIP baseline system should task image analysts with exploiting blocks of contiguous imagery by using ROIs as cues and prioritize the tasking of each block by using any secondary information in the ROI.

Each block of contiguous imagery is called an exploitable image. For spotlight-mode imagery, one whole spotlight-mode image represents an exploitable image. For search-mode imagery, one sequence of search-mode image patches is organized into a sequence of exploitable images, called screens, each of full range extent and with an equal (in pixel count) cross-range extent. The typical ground-plane extent of such an exploitable image would be around 10 km^2 for Global Hawk search mode. These considerations resulted in our defining the following requirements for the exploitation-manager design.

1. Those scenes determined by the algorithms to contain high-priority targets must be shown to an image analyst as soon as possible.
2. Each image analyst must be given sufficient visual context to make informed decisions about targets.
3. The module must keep up with the real-time data flow of incoming data.
4. All incoming imagery must be examined, at least at low resolution, by an image analyst.

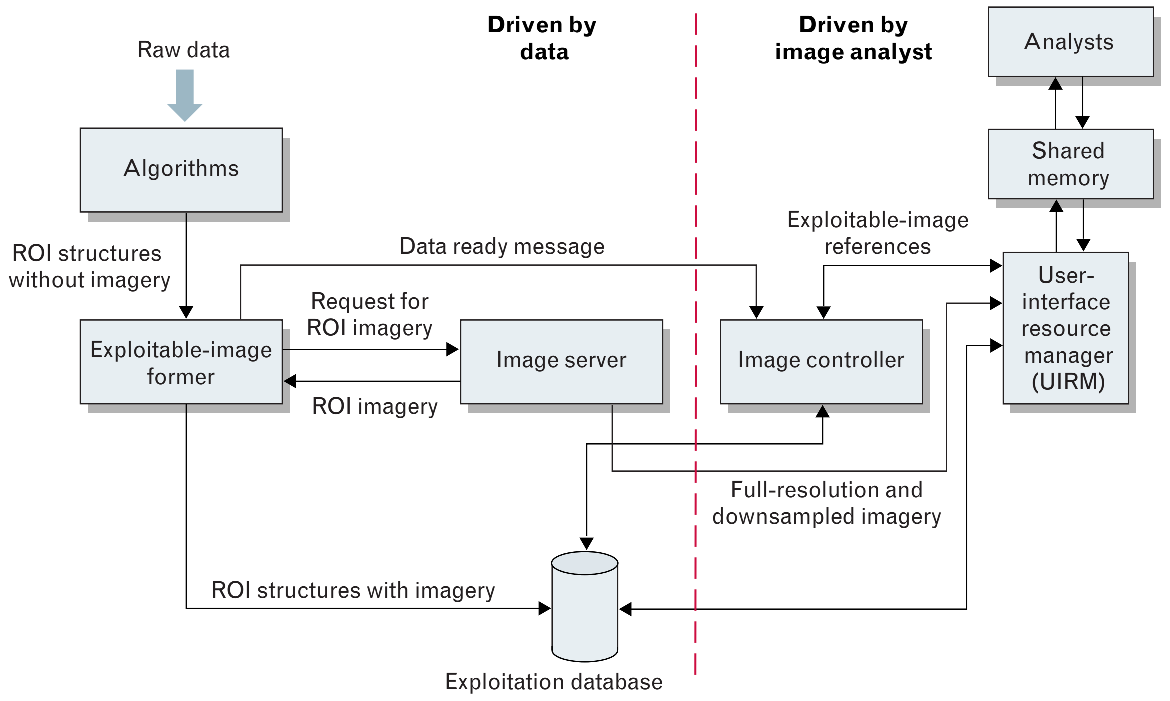


FIGURE 12. Exploitation-manager architecture. The exploitation manager is a collection of processes (exploitable-image former, router, image controller, and user-interface resource manager [UIRM]) that organize the algorithm results and imagery for display to the image analyst. The processes on the left side of the diagram must run in near real time; those on the right side respond to the needs of the image analyst.

5. If image analysts fall behind the incoming data rate, the system must buffer imagery and algorithmic results until the analysts are ready.
6. High-resolution imagery for all targets found either algorithmically or by image analysts must be saved for mission review and for later real-time retrieval.
7. Scheduling of imagery screens, subject to above considerations, is round-robin.

Data-Driven and Image-Analyst-Driven Portions of the Architecture

From these seven requirements for exploitation-manager design we see that while imagery comes in at a rate governed by the sensor, it is consumed by the image analysts at various rates based on factors such as interest of the target and expertise of the image analyst. We therefore decided to divide the system into the data-driven part and the image-analyst-driven part, as shown in Figure 12. On the left side of the figure is the data-driven part of the system, referred to as the exploitable-image former. It is image-analyst independent, and must keep up with the real-time input rate of the sensor as well as the number and size of unit detections. This data-driven portion of the system performs the following tasks: it cuts the search-mode image stream into screens according to the image boundary algorithm described below, assigns priority to each screen, stores algorithmic results with its associated context imagery in a database for post-mission and subsequent mission retrieval, and enqueues screens of imagery for the image analysts.

The data-driven segment is set into motion by receiving one of two kinds of messages from the algorithmic chain. The first type of message informs the exploitation manager that the boundary of processed imagery has moved. The second type of message delivers unit detections. The data-driven portion blocks waiting for messages from the algorithmic chain.

The image-analyst-driven portion of the system, shown on the right side of Figure 12, apportions screens of imagery to image analysts on request. They are always given higher-priority imagery before lower-priority imagery.

Search-mode imagery as it comes in from the sensor is continuous. We have to divide imagery among multiple image analysts. Requirement 2 prevents us from simply dividing the input stream into single abutting screens of data that we could enqueue on a master queue for each image analyst to pull off as needed. The force-structure-analysis portion of the algorithmic chain groups targets into units; it is considered unacceptable to cut a unit at a screen boundary, since doing so deprives the analyst of context. Thus, even if some imagery is duplicated in more than one screen of data, presenting entire units to an analyst at least once is essential. The general principal is that a unit is considered exploitable in the first screen that contains it entirely.

The above considerations determine the boundaries of screens of imagery, but not the order in which they are enqueued for review by the analyst. That order is determined by a queuing algorithm based on target priority. Each target type as well as each unit type can be categorized as high, medium, or low priority. For a given screen of data, all exploitable targets and units are checked for priority, and the highest priority of any target or unit establishes the priority for the entire screen. These priorities are used to determine the queue placement of a given screen while balancing several design criteria that include immediate presentation of high-priority imagery, reasonable load balancing among several "equivalent" image analysts, and the need to present related sequences of images to the same image analyst.

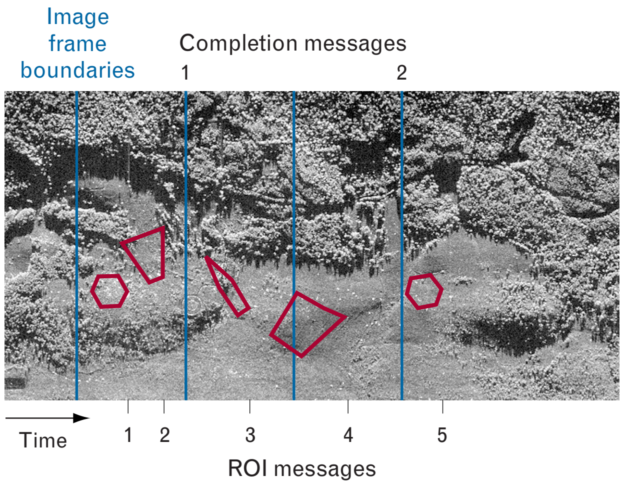


FIGURE 13. Messages received by the exploitable-image former. The sequence of messages allows the exploitable-image former to define the exploitable images and assign their priority. The completion messages guarantee that there will be no more regions of interest in frames arriving before the completion message. The ROI messages indicate that a unit has been found.

High-resolution target imagery must be stored in a database to satisfy Requirement 6. This imagery includes the entire unit as defined by the force-structure-analysis algorithm plus enough surrounding imagery (typically 0.5 km) to provide context. This requirement, however, frequently leads to a practical problem. In imagery where enough units in close proximity are present, the "padding" for the units frequently overlap. When we use a naive approach that stores all imagery for each unit and padding contiguously, overlap can cause us to store several times the total size of an exploitable image in the database. A solution to this problem is described below.

The Exploitable-Image Former

Although current systems cause whole search scenes to stream by image analysts in a continuous fashion, our system divides the imagery into manageable screens. This division allows the analyst to determine the rate at which he or she reviews the imagery, as well as allowing some critical imagery (imagery with high-priority targets) to jump the queue.

The exploitable-image former blocks until it receives one of four messages from the ATR chain. Two messages are critical for driving the system. First, the completion message contains only the frame number. It indicates that all units in the mentioned frame or before it have already been sent to the exploitable-image former—i.e., that all processing up to and including that frame has been completed and the results have already been sent so that the algorithmic results and the imagery may be released to the image analyst. Second, the ROI message indicates that a unit has been found and it contains the algorithmic results including individual targets, their classification, and the classification of the units.

Figure 13 shows a typical sequence of messages. ROI messages are emitted only when the force-structure-analysis module is certain that all targets in a unit have been seen. This event triggers two actions.

First, the necessary high-resolution imagery associated with the ROI is retrieved to be placed in the ROI database. Second, the ROI is placed into an exploitable-image structure. The message that triggers the largest amount of activity is the *Completion* message. Because it tells us that there are no new data to the left of our current position in the strip, we can emit an exploitable image whose width is equivalent to that of an image-analyst screen.

What drives the design is our earlier requirement that an image analyst should see each unit once in its entirety on a single screen. This necessitates readjusting boundaries so that a unit only partially revealed in an initial screen is guaranteed to be seen completely in a later screen.

Consider the strip illustrated in Figure 14. We draw an imaginary left boundary for the exploitable image at one screen width to the left of our current screen position. If that line cuts a unit or its padding, then we move the left boundary back to the first blank region to the left of our initial guess and place the right boundary at one screen width away from the left boundary. This repositioned exploitable image is then emitted, when possible, to the same image analyst who saw it in partial form in the previous screen. Now the unit, seen entirely, is deemed exploitable.

The story does not end there. If a right boundary cuts an ROI, the ROI is labeled unexploitable and the partial unit is shown to the image analyst for context purposes but cannot be altered. The image analyst can view the entire ROI in the next exploitable image.

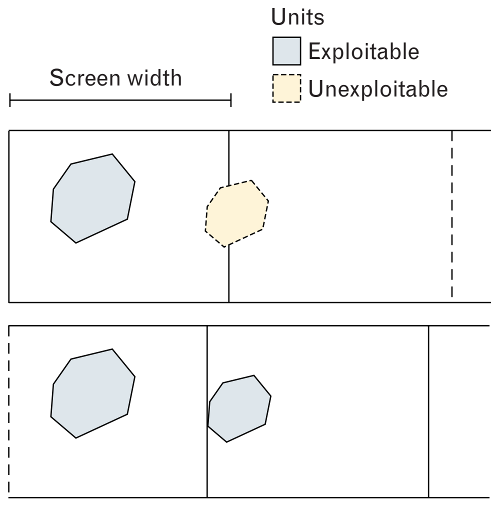


FIGURE 14. Boundary readjustment for exploitable image. A unit must always be presented entirely within an exploitable image. Boundaries are adjusted to include the entire exploitable image, even at the expense of displaying some imagery twice.

Now that the boundaries of the exploitable image have been determined, the exploitable-image priority must be set. All the units that are labeled exploitable within the image are checked. Each target type is checked to see if it is low, medium, or high priority. Then the unit types are also checked. The aggregate priority for the exploitable image is determined by taking the maximum priority for all the targets and units. Finally, the strip is divided into exploitable images that are stored in a database and then sent to a queue managing process.

Once an exploitable image is formed, it includes both image boundaries and a list of force units that are enclosed within the boundary. Some force units within an exploitable image are totally enclosed within the boundaries of the image, and some straddle boundaries. High-resolution imagery, including padding to provide context, must be saved for each unit as well as for the algorithmic results. The issue of providing context padding for each chunk of ROI imagery leads to a complication. Certain exploitable images are rich in units. The padding of each unit to include 0.5 km on each side can lead to significant overlap in imagery, as shown in Figure 15.

Our solution involves the use of tiles. A tile size, typically 256 × 256 pixels, is chosen and an exploitable image is divided into a grid of tiles, as in Figure 15. The first ROI is stored with all its tiles in the database and the ROI is given a list of pointers to its tiles. For each successive ROI that is stored, the database is checked on a tile-by-tile basis to see if that tile has already been stored and cached. If it has, the new ROI is given a pointer to the existing tile for its list. Otherwise, a new tile is stored and a pointer to it is saved with the ROI. This arrangement guarantees that the amount of high-resolution imagery stored cannot exceed the size of the exploitable images.

The database stores (1) the frame objects for every frame completed by the force-structure-analysis module; (2) tile objects for imagery covering a unit; (3) ROI objects containing algorithmic results for a unit; and (4) exploitable images that contain the bound-

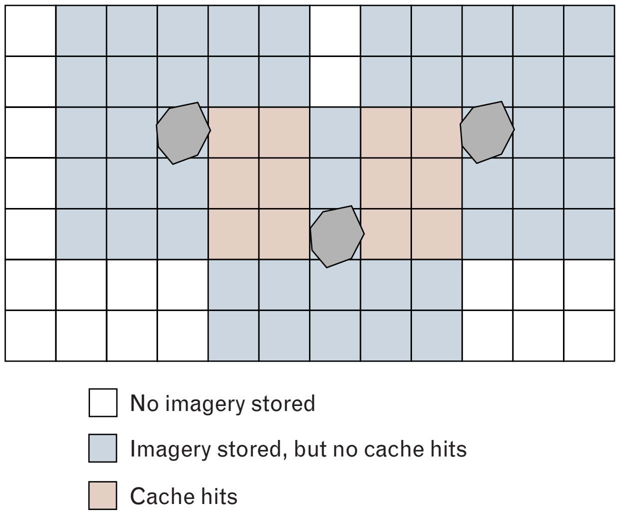


FIGURE 15. Tile storage and caching scheme for ROI imagery. All image tiles surrounding the ROIs at the left and right are stored and cached. By the time the center ROI is processed, many image tiles have already been stored.

aries of the exploitable image and pointers to all the ROIs contained by it.

The data needed to construct the above objects are emitted asynchronously from the force-structure-analysis module. While frame objects come in at a relatively steady rate, depending only on the radar mode, the remaining objects are completely data dependent, and form a size-based hierarchy—several tiles are needed to make up a unit and from zero to many units will be contained within an exploitable image. Resource utilization becomes an issue. We can wait until the end of an exploitable image before we store anything in the database, but doing so causes resources like data channels to imagery and access to the database to go fallow until the end of an exploitable image, at which time large amounts of data will have to be transported from the image feed and to the database on disk. Therefore, a scheme was devised to put objects into a database as soon as they can be formed, and to put container objects into a database with references to objects already in a database.

An in-memory cache of pointers for already seen tiles is kept for quickly retrieving references for tiles for new incoming ROIs. Each frame is added to the database as it comes in. As an ROI is emitted, the tile cache is checked to see if any new tiles need to be retrieved and put into the database and added to the cache table. The ROI itself, including the algorithmic results, is then added to the database and placed on a list of ROIs to be added to the exploitable image, which is then formed when a frame completion message indicates that we have reached the required right-hand boundary.

Implicit in the description above is the idea that sometimes an object resides in local memory (referred to as transient memory), in database storage (referred to as persistent memory), or in both. Typical classes contain C-style pointers that work only with memory references. We find it useful to have algorithms work in such a way that an object in memory is accessed directly and an object in the database is retrieved. Furthermore, if a copy is made, necessary reference counts to both the memory and database versions should be added independently. This encapsulation of the data-referencing function and the copying function is achieved through the use of specially designed smart pointers that keep references to both the transient and/or persistent copies of data. Issues such as accessing database data only within transactions are also hidden within these smart pointers, which form a uniform interface to the database.

Exploitable images are queued for the analysts. The design of the queuing manager was based on the following assumptions. First, exploitable images exist in three priorities (high, medium, and low). High-priority images are always handled before medium-priority images, which are always handled before low priority images. Second, all image analysts are equally capable of handling any imagery, and all other things being equal, the system will attempt to distribute the load among the analysts. Third, we give each image analyst analyzing a sequence of nearby targets as much context as possible.

If we had only one priority for exploitable images, the solution would be well known. We would have a single queue from which all three analysts would remove nodes. Provided that a semaphore protects the end of the queue against a software component monopolizing the system, we could satisfy all three of the above assumptions. Priorities could be added by maintaining three separate queues, and each consumer could consume from the highest-priority queue with elements. This arrangement violates the

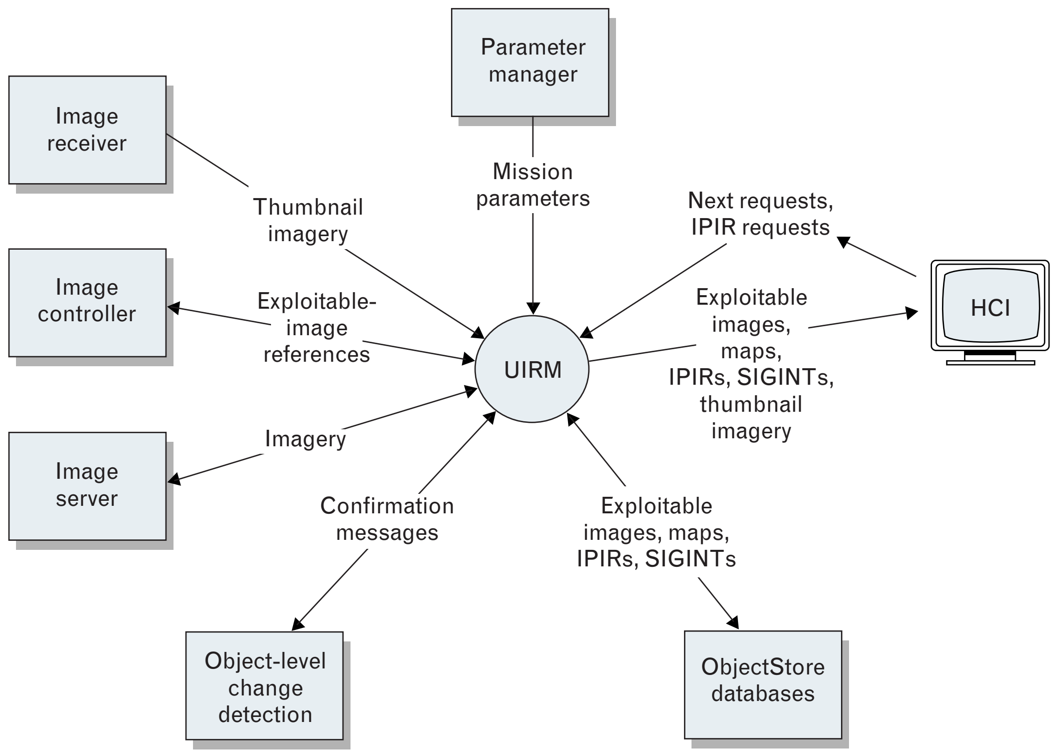

Parameter manager
Mission parameters
Image receiver
Thumbnail imagery
Image controller
Exploitable-image references
Image server
Imagery
UIRM
Next requests, IPIR requests
HCI
Exploitable images, maps, IPIRs, SIGINTs, thumbnail imagery
Confirmation messages
Exploitable images, maps, IPIRs, SIGINTs
Object-level change detection
ObjectStore databases

FIGURE 16. The user-interface resource manager (UIRM) external interfaces. The UIRM must coordinate many kinds of data for display to the image analyst, and store the analysts' results. The UIRM uses four types of ObjectStore databases: map, signal intelligence report (SIGINT), initial phase interpretation report (IPIR), and ROI. The SIGINT database maintains signal intelligence reports, and the IPIR database maintains initial phase interpretation reports.

third assumption involving context. If an image analyst first sees only part of a unit and it is deemed unexploitable, we want to send the entire unit to the same image analyst as part of the next exploitable image. If we simply place the entire unit on a common queue, there is no guarantee that it will go to the same image analyst. We could have linked the subunit and unit, but we did not want to complicate the queue data structures. Therefore, each image analyst was given a triplet of queues, and exploitable images were placed on each queue in a round-robin fashion. It should be noted that the nodes themselves contain only image boundaries and database references to algorithmic results, but not imagery. Therefore, the queues can grow quite long without exhausting resources.

User-Interface Resource Manager

The user-interface resource manager (UIRM) acts as an interface between an X-Windows-based component called the human-computer interface (HCI) and the rest of the SAIP system. The HCI presents results to the user and responds to user requests. To service these requests, the HCI must obtain data from various external processes with minimal interference and delay for the user. The HCI and the UIRM are threads within the same process; they communicate via shared memory and pipes. These mechanisms are similar to those used in algorithm components, so that the different threads operate as independently as possible to minimize any delays in user interactions with the screen. Figure 16 shows the UIRM with its many external interfaces.

The UIRM uses four types of ObjectStore [10] databases: map, signal intelligence report (SIGINT), initial phase interpretation report (IPIR), and ROI. The SIGINT database maintains tactical signal intelligence reports, and the IPIR database maintains IPIRs. The database that the UIRM interfaces with most frequently is the ROI database. It contains exploitable images and their associated ROIs.

Figure 17 shows how the UIRM process is composed of four threads—HCI, IPC, UIRM input, and UIRM output—that communicate via shared memory and pipes. The HCI thread is an X-Windows-based task that responds to image-analyst inputs. The IPC thread, which is created automatically when the SAIP-IPC library is initialized, is responsible for the transfer of data between the UIRM process and its external processes. Because the major function of the UIRM is to service HCI requests for external data, the UIRM input and output threads form the greater part of the UIRM process.

The input thread is responsible for processing messages received, via SAIP IPC, from external processes. It is activated by the IPC thread when a message is received for thumbnail imagery (highly downsampled imagery), a reference to an exploitable image, downsampled imagery, or full-resolution imagery. Once the message is deciphered, the input thread notifies either the output thread or the HCI thread, via a pipe, that data have been received.

The output thread is responsible for servicing requests from the HCI thread. It sends requests for data to external processes, and when notified by the input thread that data has been received, it notifies the HCI thread. The output thread is also responsible for all database access for the UIRM process. The output thread is activated by receipt of a message on the pipe from the HCI or input threads. The primary message that it receives from the HCI thread is the Next request, which is discussed below. The other two types of requests that the UIRM can receive from the HCI are related to IPIRs.

With the exception of the image receiver, communication between the UIRM and the rest of its external processes is triggered by requests from the HCI component. The UIRM receives unsolicited thumbnail imagery from the image receiver at regular intervals. This section describes the requests that the UIRM can receive from the HCI, and explains the communication that results between the UIRM and its external processes in order to service these requests.

The UIRM receives HCI requests for data via a pipe. A pipe is used because the requests are only a few bytes and the pipe allows multiple requests to be

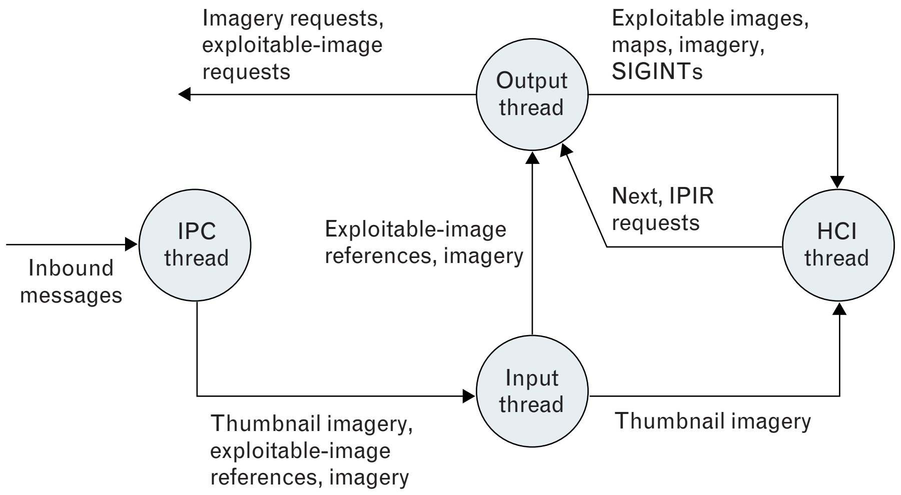

Imagery requests, exploitable-image requests
Exploitable images, maps, imagery, SIGINTs
Output thread
Next, IPIR requests
IPC thread
Exploitable-image references, imagery
HCI thread
Inbound messages
Input thread
Thumbnail imagery, exploitable-image references, imagery
Thumbnail imagery

FIGURE 17. UIRM architecture diagram. The UIRM process has four threads to coordinate movement of data between databases and the human-computer interface (HCI). The UIRM input thread is responsible for processing messages received, via SAIP IPC, from external processes. The output thread is responsible for servicing requests from the HCI thread, primarily the Next request, and maintaining all database access for the UIRM process. The HCI thread is an X-Windows-based task that responds to image-analyst inputs. The IPC thread, which is created automatically when the SAIP IPC library is initialized, is responsible for the transfer of data between the UIRM process and its external processes.

queued correctly. The UIRM obtains the necessary data either from external processes or from ObjectStore databases. The UIRM stores the obtained data in shared memory and notifies the HCI, via the pipe, that it is available. The UIRM can receive three requests from the HCI via a pipe:

1. Next request—the image analyst presses a button on the HCI screen to view the next exploitable image. This action triggers the UIRM to perform a series of events that ultimately result in a delivery of information to the HCI in an efficient manner.
2. A request to submit an IPIR—this request is issued by the HCI whenever the image analyst generates an IPIR. In response to this request, the UIRM obtains the IPIR from shared memory, and inserts it into the IPIR database.
3. A request for a default IPIR—in response to this request, the UIRM retrieves the default IPIR from the database, stores it in shared memory, and notifies the HCI, via the pipe, that the default IPIR is available.

The UIRM can be characterized as a complex data router. Its actions, in response to the press of the Next button, are summarized below.

1. Any modifications, additions, or deletions made by the image analyst to existing ROIs within an exploitable image are incorporated into the ROI database. A message to the object-level change-detection process confirming the nature of the detections in the exploitable image is generated.
2. The UIRM sends a request for an exploitable image from the ROI database to the image-controller process. Some data are double buffered (i.e., loaded before they are needed) in the shared-memory region to improve response time to the image analyst. The imagery associated with each ROI in an exploitable image is not extracted from the database because the UIRM obtains the complete image from the image server. This arrangement improves latency. Each ROI contains a list of previously seen ROIs that overlap this one so that the UIRM cycles through the array of ROIs associated with the exploitable image and extracts the most recent old ROI associated with each. The UIRM also requests downsampled imagery (about 1 MB) from the image-server process.
3. The UIRM also sends a message to the HCI to indicate that the double-buffered data (i.e., the exploitable image and the downsampled imagery) are available to allow the image analyst to begin exploitation. Meanwhile, there are more data to be loaded.
4. The UIRM requests full-resolution imagery from the image server. This request takes a long time to be fulfilled. The UIRM does not wait for it to be completed, but begins to obtain additional data required by the HCI. It obtains the geographic boundary from the exploitable image, and uses this information to extract appropriate SIGINT reports from the SIGINT database and maps from the map databases.
5. When the full-resolution imagery is finally received from the image-server process, the UIRM notifies the HCI that the Next-request processing is complete. The image analyst now has a new exploitable image to interpret.

Initial Field Experiences

The complete SAIP hardware and software system was assembled by the contractor team at Lincoln Laboratory and then transferred by Northrop Grumman to a militarized van similar to the one used for the Enhanced Tactical Radar Correlation (ETRAC). In March 1997 the complete system was used to observe the Task Force XXI exercise [11] at the National Training Center, Fort Irwin, California. It was also used the following month to observe the Roving Sands exercise in New Mexico [12]. Both tests used the Advanced Synthetic Aperture Radar System-2 (ASARS-2) sensor on the U2 aircraft. Table 5 summarizes the test experience in New Mexico from the system-reliability point of view.

The system operated reliably for all of the missions and well within its throughput capabilities. The general architecture of the system seemed well suited to the task, and the image analysts seemed pleased with the general capabilities of this prototype system. Northrop Grumman continued development with support of all the contractors to produce enhanced versions of the system.

Table 5. SAIP System Flight Test Experience in New Mexico

Flight Date in 1997	*Collection Time (hr)*	*Total Image Batches*
14 March	4	244
26 March	3	287
26 March	3	317
27 March	3.75	196
27 March	3	151
21 April	3.5	142
22 April	3.5	192
Total	23.75	1529

Acknowledgments

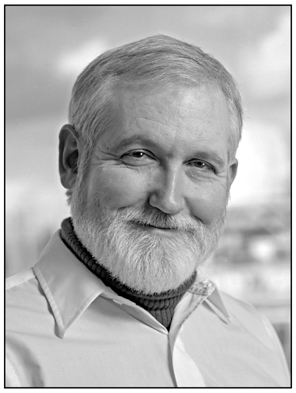
The image-registration components of SAIP are supplied by Harris Corporation. The terrain delimitation, cultural-clutter identification, spatial cluster, and force-structure-analysis components are supplied by Oasis Research Center. The object-level change-detection component is supplied by Atlantic Aerospace Electronics Corporation.

We would like to acknowledge the many contributors to the design and development of SAIP here at Lincoln Laboratory and other companies. Of special mention are Sandra Crocker, who led the design of the baseline SAIP human-computer interface, the destination of all the results for the software described in the article; Michael Pagels of Oasis Research Center; Paul Baim and Theodore Bially of Atlantic Aerospace Electronics; and Thomas Kerr and Jay Hackett of Harris Corporation for detailed design discussions.

REFERENCES

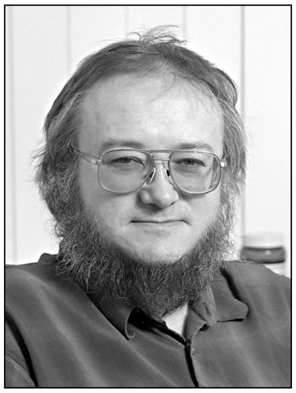
1. "HAE UAV Concept of Operations," DARPA, 1995.
2. M.T. Fennell, B.B. Gragg, and J.E. McCarthy, "Analysis of Advanced Surveillance Architectures," Toyon Research Corporation, Dec. 1994.
3. C. Lam, "Exploitation Considerations for the High Altitude Endurance Unmanned Aerial Vehicle (HAE UAV) System," DARPA, Aug. 1994.
4. R. Bates-Marsett, "CONOPS Outline for Imagery Analysis Function," working document from MRJ Corp., Jan. 1995.
5. R.J. Vetter, "ATM Concepts, Architectures, and Protocols," *Commun. ACM* **38** (2), 1995, pp. 30–38, 109.
6. S.C. Crocker, D.E. Kreithen, L.M. Novak, and L.K. Sisterson, "A Comparative Analysis of ATR Operating Parameters and Processing Techniques," *MIT Lincoln Laboratory Report MS-10323,* published in *39th Annual Tri-Service Radar Symposium, June 1993.*
7. A.A. Stepanov and M. Lee, "The Standard Template Library," *Technical Report HPL-94-34,* Apr. 1994, revised 7 July 1995.
8. M. Nelson, *C++ Programmer's Guide to the Standard Template Library* (IDG Books, Foster, Calif., 1995).
9. L.M. Novak, G.J. Owirka, W.S. Brower, and A.L. Weaver, "The Automatic Target-Recognition System in SAIP," *Linc. Lab. J.* **10** (2), 1997, pp. 187–203.
10. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *Commun. ACM* **34** (10), 1991, pp. 50–63.
11. See the Internet web page http://call.army.mil/call/exfor/ted/toc.htm
12. See the Internet web page http://www.forscom.army.mil/Rsands/overview.htm

L. KEITH SISTERSON
investigates and develops systems for synthetic aperture radar (SAR) and moving target indicator (MTI) data acquisition, processing, and exploitation as a staff member of the Surveillance Systems group. He joined Lincoln Laboratory in 1989, after managing teams in the private sector to build software for microprocessor development, materials testing, and real-time hospital information systems. Before that, he researched experimental high-energy nuclear physics at Harvard University. Keith holds an M.A. degree in natural sciences from the University of Cambridge in England, and a Ph.D. in high-energy nuclear physics from Imperial College of Science and Technology in London. He belongs to the American Physical Society and the Association for Computing Machinery.



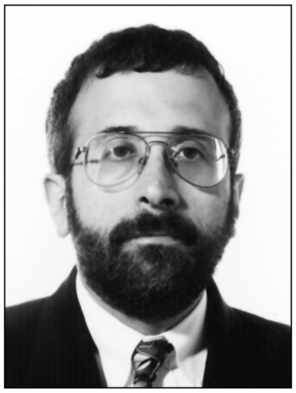
JOHN R. DELANEY
is a staff member of the Systems Analysis group. He performs systems analysis and designs software for underground facilities. John has also worked in systems analysis and software design for Alphatech in Burlington, Massachusetts. He holds B.S., M.S., and Ph.D. degrees in electrical engineering from Stanford University in California.

SAMUEL J. GRAVINA
is a staff member in the Surveillance Systems group, developing real-time image-exploitation algorithms and software. Before joining Lincoln Laboratory in 1996, Sam was a research consultant with the Department of Radiology Research at Beth Israel Hospital, Harvard Medical School, where he conducted structural and chemical magnetic resonance imaging (MRI) studies of cartilage degradation in osteoarthritis. From 1990 to 1995 he worked as an applications scientist at Bruker Instruments, a scientific instruments manufacturer. There he developed software, hardware, and research techniques for use in a wide variety of problems including microscopic MRI, high-resolution imaging of solid materials, real-time image acquisition in dynamic biological and physical systems, imaging of human cognitive functioning, and multidimensional image analysis and display. From 1988 to 1990, Sam worked as a postdoctoral research associate at Rensselaer Polytechnic Institute, studying the magnetic and electrical properties of novel high-temperature superconductors. Sam holds a B.S. degree in physics from Rensselaer Polytechnic Institute in Troy, New York, and M.S. and Ph.D. degrees in physics from Brown University in Providence, Rhode Island. He is a member of the American Physical Society.



PAUL R. HARMON
is a staff member in the Sensor Exploitation group. His work in software engineering focuses on parallel software systems design and implementation. He joined the Laboratory in 1985, after working with network system software for the Canadian Educational Microcomputer Corporation (CEMCORP). He holds B.A.Sc. and M.A.Sc. degrees in electrical engineering from the University of Toronto in Ontario, Canada.

MARGARITA HIETT
is an associate member of the Radar Imaging Techniques group. She joined the Laboratory in 1993, and her current research work involves analyzing satellite images. From 1986 to 1992 she held a technical staff position at Raytheon, where she was involved in the implementation of large-scale real-time hardware and software systems for detection and discrimination of infrared (IR) targets. From 1992 to 1993 she worked at Textron, where she continued her work analyzing real-time algorithms for detection of moving IR targets. Margarita received a B.S. degree in electrical engineering from the University of Massachusetts at Amherst.



DANIEL WYSCHOGROD
is a former associate staff member of the Information Systems Technology group. He now designs a risk management database system for BankBoston's Treasury Systems division. Dan worked at the Laboratory from 1989 to 1998. He also worked in the Air Traffic Surveillance group, designing and implementing a detection and tracking algorithm for an airport surface surveillance system that was subsequently licensed to the Raytheon Corp. and incorporated into a surveillance system provided to the government of India. Dan also developed a Perl-based system for detecting illegal transitions to root from recorded session transcripts. Before coming to the Laboratory, he worked on optics and image-processing projects at Sparta Systems, and on vision-guided robotics projects at Draper Laboratories. He holds a B.A. degree in physics from Columbia University in New York; an M.A. degree in physics from the State University of New York (SUNY) at Stony Brook; and an M.S. degree in computer science from the Courant Institute of New York University.