# Management of Sensori-Motor Activity in Mobile Robots

Azer A. Bestavros James J. Clark Nicola J. Ferrier

Division of Applied Sciences
Harvard University
Cambridge, MA

**Abstract**

A mobile robot system will typically have associated with it a number of sensing subsystems. If these sensing subsystems are "active", that is, using motion to aid in the sensing process [1, 2, 8, 9, 14], they will each be issuing motor requests related to the sensory processing algorithm that they are performing. In addition, the manipulatory portions of the robot system (i.e. those which uses its arms, grippers and so on, to manipulate objects) will also be requesting motions. Thus an important consideration in the design of sensori-motor systems in robotics is the management of conflicting motion requests from multiple sensory and manipulatory modules.

This paper examines the motion management problem and we present in this paper an architecture, based on Bestavros' IOTA abstraction [5], that allows one to schedule the motor commands sent to the various actuators in the robot in a manner appropriate to the goals of the robot.

## 1 Introduction

In a general mobile robot the motor systems of the robot may be required to perform duties related both to the acquisition of sensory information (i.e. *active sensing* [1, 2, 8, 9, 14]), as well as to influencing of the robot's environment. For example the active sensing system may require the robot to move around a block in order to see what is behind it, while the manipulative system may need the robot to stand still so that it can grasp a nearby object which would be out of reach if the robot moved. It can be seen by this simple example that one cannot decouple the motor activities required by the sensing system from the motor activities required by the manipulative system. Thus any system that is developed for controlling motor activities in a robot must take into account both the perceptual goals and the manipulative goals of the machine and produce motions which address these goals in an integrated and orderly manner.

One can think of the motor units of the robot as a limited resource that must be shared between the sensory and manipulative systems. The motion control operating system must arbitrate between the conflicting requirements of these two systems in a way which allows the goals of the two systems to be attained.

**This research was supported in part by the ARO Brown/Harvard/MIT Center for Intelligent Control Systems under grant DAA103-86-K-0171, by the Harvard/University of Maryland NSF Systems Research Center under grant number CDR-85-00108, and by DARPA grant N00039-88-C-0163.**

## 2 An Operating System for Sensori-motor Control

A mobile robot will typically have a number of goals that it is trying to achieve at a given moment in time. The attainment of these goals require the robot to undertake a number of tasks. In general, however, a robot will only be able to perform one task at a time. The tasks must therefore be prioritized according to the relative importance of the goals these tasks subserve, and the most important is then worked on. The relative importances of the tasks may change with time, so that a given task may be interrupted by a new, now more important, task. In addition, a given task may have temporal constraints associated with it. For example, a task may require execution within a certain time since its request, after which it becomes obsolete or invalid. Or a task may be guaranteed to complete in a specified interval since the beginning of its execution. The motion management system must take these into account when determining what motion to undertake.

These considerations lead us to the idea that a mobile robot requires a multi-tasking, interrupt driven, planning system for determining which motion it is to execute at a given time. This planning system would examine the robots current estimation of its environment, based on its current sensory input, and determine which task it is to work on. If this requires that a given task be interrupted, a mechanism must exist that will allow resumption of the interrupted task once the interrupting task has been completed. This means that the state of a given sensory routine be saved upon interruption so that it can be resumed properly. Again, one can make a direct analogy between the operation of this system and an interrupt driven multi-tasking computer.

We propose to base our sensori-motor operating system on the MDL (Motion Description Language) developed by Brockett [3, 6, 10] and on the concept of I/O Timed Automata [5]. The MDL language provides a device independent mechanism for the specification of the motions of robotic systems. The I/O Timed Automata allows us to construct a sensori-motor operating system that meets the requirements described above.

In our proposed system motion requests from the sensory and manipulative systems are presented to the operating system in the format of MDL programs. The sensori-motor operating systems then processes these requests with reference to the goals of the robot and all relevant physical or temporal constraints, and passes on suitable motor commands, expressed in MDL format,

to the individual motor controllers for each degree of freedom possessed by the robot. Detailed descriptions of the MDL approach to robot control can be found in [6, 10]. The I/O Timed Automata, and its application to robotics, is described in the following sections.

# 3 The $\Im$ Framework

The current practice in building real-time embedded systems is not based on any sound scientific approach [13]. In view of the increasing complexity, cost and criticality of these systems, it has become evident that a new methodology should be adopted in their design and implementation.

In [5], the Input-Output Timed Automata (IOTA or simply $\Im$ – read "yota") model was introduced. The $\Im$ model is expressive. It offers a clean methodology for incorporating timing constraints in the problem specification. Its asynchronous non-blocking (reactive) nature allows accurate and realistic specification of real-time systems. In [4], we have demonstrated how $\Im$'s can be "naturally" used for specifying different activities in a wide range of real-time applications (*e.g.* robotics, neural networks, asynchronous circuits, ... *etc.*). The use of a high level programming language based on that model within an environment that would provide the necessary support for the development cycle of real-time applications was proposed. In this section, we overview the $\Im$ model and discuss its potential use as a framework for the specification of integrated sensory and manipulative activities.

## 3.1 The $\Im$ Model: An overview

Using the $\Im$ model of computation, a system is viewed as a set of interacting automata called $\Im$'s (IOTAs). Each one of these $\Im$'s represents an entity in the system. This entity might be a software module,[1] a system resource,[2] or a hardware component.[3]

$\Im$'s communicate with each other and with the external environment through *channels*. A channel is an abstraction that encapsulates the notion of a communication media. Among others, a channel might represent a simple physical wire, a Unix-like socket or pipe, a function invocation, or, a possible transfer of information through function calls and returns. The information that a channel carries is called a *signal*. A signal consists of a sequence of *events*. An event underscores the instantiation of an *action* at a specific point in time.

A channel changes the value of its signal by firing a new action. Once this happens, the computation associated with that action is performed. This might result in a state transition. The firing of an action, along with the necessary computation and state transition, are assumed to be done *atomically*: that is *indivisibly* and *irreversibly*.

[1] *e.g.* a scheduler, a resource manager, a motion control algorithm, ...*etc*

[2] *e.g.* a communication network, a shared memory, a special purpose image processor, ...*etc*

[3] *e.g.* an actuator, a sensor, ...*etc.*

Channels, and consequently the signals and actions that they carry, are classified as *input* or *output*. Signals from input channels are uncontrollable since they are supplied by the environment or by other $\Im$'s. Unlike other models [12], they cannot be blocked. This notion is crucial for the modeling of real-time systems, where state transitions are forced by the environment. Signals from output channels, on the other hand, are the way a $\Im$ reacts to the outside stimuli. They are enabled only when the $\Im$ is in one of some specific set of states. Note that the specific action that would fire, is state dependent. Thus, the state of a $\Im$ determines which channel(s) are enabled and which action(s) are to fire. Input and output channels are called *external* because they can be observed from outside a $\Im$. Channels used to signal actions that are observable only locally are called *internal*.

To reflect timing constraints, lower and upper time bounds are associated with each local channel to determine when its signal should change its value (by firing an action) if it ever becomes enabled. For instance assume that $\tau_l^c$ and $\tau_u^c$ are the lower and upper time bounds associated with a local channel $c$. Furthermore, assume that $c$ became enabled at time $t_i$, then $c$ can fire at any time $t_j \in \{t_i + \tau_l^c, t_i + \tau_u^c\}$ provided that it remained enabled during the time interval $\{t_i, t_j\}$. In a sense, these time bounds define a timing constraint on the response of the $\Im$ to some conditions.

A specification of a $\Im$ is a description of its behavior (*i.e.* how it reacts to stimuli from the environment). A $\Im$ is said to *implement* another $\Im$, if it is impossible to differentiate between their external behaviors. This is the primary tool that is used to verify that an implementation meets the required specification. Finally, to allow for modular and hierarchical specification, $\Im$'s can be *composed* together to form new $\Im$'s.

## 3.2 The $\Im$ Model: An Example

Consider a robotics system designed to manipulate objects moving on a conveyor belt using a gripper mounted on the end-effector of a six-degrees of freedom arm. A camera system mounted on the same end-effector is used to locate the objects and to monitor the surrounding environment. Figure 1 illustrates the different components in such a system. The planning process accepts various sensory information and generates perceptual (in this case visual) and manipulative goals accordingly. To meet these goals, the vision control and the manipulative control processes might request different end-effector positions. These possibly conflicting requests are handled by a motion scheduler which determines the actions to be taken by issuing MDL commands [6]. These commands are processed by an MDL interpreter which drives the motor controllers.

Using the $\Im$ framework, each one of these processes would be specified by a $\Im$. For instance, consider the specification in Figure 2 of a simple motion scheduler. It accepts request signals from both the vision control and the manipulative control $\Im$'s. Each request consists of an MDL[4] command [6], a priority level and a time bound. The MDL command consists of a triplet $< u, k, t >$, where $u$ defines the control inputs (or setpoints), $k$ determines the effect of the feedback vector, and $t$ is the length of time (or epoch) the command should be applied. The scheduler response to the

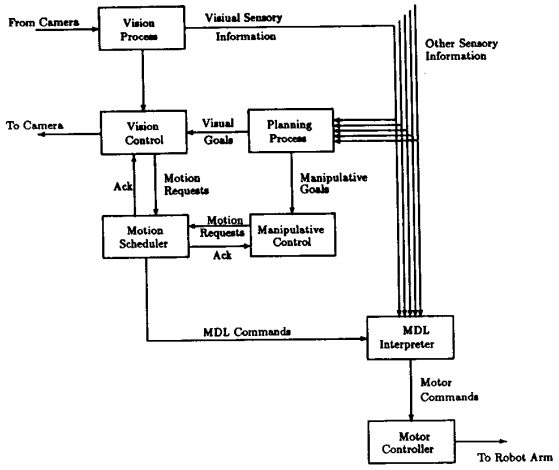



Figure 1: An example of a robotic sensori-motor system

requester can be either **COMMIT**, **ABORT**, or **REJECT**. The **COMMIT** response means that the request was completed successfully. The **ABORT** response means that the request had to be aborted in favor of a higher priority request. The **REJECT** response means that the request cannot be serviced due to a conflict with a higher priority request.

In Figure 2 we use a $\Im$-based specification language, ESPRIT,[5] to describe the motion scheduler. The specification consists of two parts: an *interface* and a *body*. In the interface, the name of the $\Im$, its external input and output channels along with the range and switching time of these channels are specified. The interface provides all the information needed by other $\Im$'s to interact with the scheduler. In particular, it provides enough information for type checking and compatability of signals. For instance, our scheduler accepts inputs from N channels `req[0]`, `req[1]`, ..., `req[N-1]`. The values expected on these channels are specified by the *mdl_req* structure. Furthermore, the minimum time between two requests on anyone of these channels is 1 time unit. The specification of the outputs of the scheduler can be explained in a similar way.

The body of a $\Im$ identifies its behavior. It consists of three parts: the *declaration part*, the *invocation part*, and the *computation part*. In the declaration part, the state variables, their ranges and their initial values are given; the internal channels are specified; and, local $\Im$'s are identified. In the inclusion part, local $\Im$'s are instantiated and their input/output channels are bound[6]. For instance, our scheduler has three state variables: the `request` vector and the `current_master` and `current_priority` variables. The `schedule[i]`, `commit[i]`, and `reject[i]` channels are internal – they are invisible outside the body of the $\Im$. Our simple scheduler is not defined in terms of any other $\Im$'s, thus, there are no included $\Im$'s in its body. In the computation part, the behavior of the $\Im$ with respect to each of its actions is specified. This is done using three clauses. The *enable clause* specifies a pre-condition on the state of the $\Im$. Whenever this pre-condition becomes true, the associated action becomes

[4]Motion Description Language
[5]Executable Specification of Parallel Real-time Interactive Tasks
[6]assigned to output/input signals of other $\Im$'s

```
struct mode {
  float w, k ;
} ;

struct mdl_req {
  struct mode control ;
  float epoch, timeout ;
  int priority ;
} ;

struct request {
  int status ;
  struct mdl_req command ;
} ;

iota scheduler:  req[N](struct mdl_req)/1 → ack[N](int)/1, mdl-command(struct mode)/1
  {
    states
      struct request request[N] = NO_REQUEST ;
      int current_master = NULL, current_priority = 0 ;

    channels
      schedule[N]()/1, commit[N]()/1, reject[N]()/1 ;

    actions (i=0;i<N;i++)
      req[i](x)
        do { request[i].status = PENDING ; request[i].command = x ; }

      schedule[i]()
        when (request[i].status == PENDING ∧
              request[i].command.priority > current_priority)
        do {
          if (current.status == SIGNALING)
            request[master].status = ABORTED ;
          request[i].status = STARTING ;
          current_master = i ;
          current_priority = request[i].command.priority ;
        }

      commit[i]()
        when (request[i].status == SIGNALING)
        after request[i].command.epoch
        do {
          request[i].status = COMMITTED ;
          current_master = NULL ;
          current_priority = 0 ;
        }

      reject[i]()
        when (request[i].status == PENDING ∨ request[i].status == STARTING)
        after request[i].timeout
        do { request[i].status = REJECTED }

      mdl-command(request[current_master].command.mode)
        when (request[current_master].status == STARTING)
        do { request[current_master].status = SIGNALING }

      ack[i](ABORT)
        when (request[i].status == ABORTED)
        within [0,T]
        do { request[i].status = NULL }

      ack[i](COMMIT)
        when (request[i].status == COMMITTED)
        within [0,T]
        do { request[i].status = NULL }

      ack[i](REJECT)
        when (request[i].status == REJECTED)
        within [0,T]
        do { request[i].status = NULL }
  }
```

Figure 2: $\Im$ specification of a motion scheduler

enabled. The *time-constraint clause* determines a time interval within which the action should be taken in case it becomes and remains enabled. Input actions (for example `req[i](x)`) cannot be blocked, thus, they cannot be associated with enable or time-constraint clauses. The *do clause* specifies how the state variables should be affected by the firing of the associated action.

We illustrate how actions are taken, by looking at our simple scheduler once more. It should commit a request if it were able to signal the appropriate MDL controls for the required **epoch** of time. This is what the `commit[i]()` internal action is expressing. It fires if the requested **mode** is signaled for an amount of time equal to the requested **epoch**. Committing a request results in the scheduler moving to a state where it is free to accept a new (or pending) request. This is depicted in the do clause of the `commit[i]()` action. The `commit[i]()` action is an example of an action that is associated with a "hard" time constraint. The `ack[i](x)` output actions, on the other hand, are examples of

actions associated with "soft" time constraints where the action is allowed to fire within an interval of time. The absence of the time-constraint clause in an action specification means that it is allowed to fire at any time as long as its pre-condition holds. The `schedule[i]()` internal action is an example of "unconstrained" actions.

The motion scheduler that we used as an example to introduce the $\Im$ model could have been specified in very different ways. For example, we made an unstated assumption that if sensory systems $S_1$ and $S_2$ submit requests $R_1$ and $R_2$ respectively, where $R_1$ is optimal with respect to $S_1$ and $R_2$ is optimal with respect to $S_2$ and the timing constraints are such that only one of the requests can be executed, then one of these requests will be granted based (say) on some priority scheme, whereas the other will be rejected. There are ways, however, to suboptimally satisfy both requests. For instance, if instead of submitting a request for (say) a position vector, $S_1$ and $S_2$ also provide the scheduler with some cost function (how costly is it to be in position $P+dP$ instead of position $P$), then it is indeed possible for the scheduler to minimize the total cost by issuing a move command to an intermediate position. The cost function will depend on the degradation of the expected sensory information and on the relative importance of that information to the overall system operation at that point in time. The expression of the above *fusion* ideas using $\Im$'s should be straightforward. This is not the case with methodologies like Brooks' sumbsumption architecture, [7], where (as it will be explained in the next section), even a dynamic priority scheme cannot be supported.

### 3.3 Behavioral Specification using the $\Im$ model

Building autonomous creatures has been a particularly interesting and challenging area in robotics research. This goal has led to research in building control systems based on "task-achieving behaviors" [11], that can deal with multiple goals and multiple sensory information. In this section, we show how the $\Im$ model (and language) can be used to specify such behaviors in a natural, concise and elegant way. We contrast the $\Im$ framework with Brooks' subsumption architecture [7]. In particular, we show that the $\Im$ model is more general and more expressive; it "subsumes" the subsumption architecture.

In [7], Brooks proposes the *subsumption architecture* as a methodology for specifying and building complex control systems. This architecture suggests the use of a vertical decomposition of the control system into a number of parallel independent task-achieving behaviors. Each one of these layers of behaviors is made out of smaller units called *modules*. Each module has a *finite* state controller and a certain number of inputs and ouputs for communicating with other modules. There are two distinguished inputs for each module *reset* and *inhibit*. Reset forces the finite state controller to go back to its initial state. Inhibit prevents the module from producing its output. Another special form of interaction, the subsumption, allows a module from a higher layer to overwrite the output of a module from a lower layer. The higher layer is called a dominant behavior, whereas the lower layer is called an inferior behavior. Subsumption allows control systems to be patched up by allowing smarter (or higher priority) behaviors to take over from default behaviors whenever appropriate.

The subsumption architecture can be supported easily and effectively using the $\Im$ model. A module is simply a $\Im$ with its inputs (outputs) being the input (output) signals of the $\Im$. The reset and inhibit inputs can be easily implemented as always-enabled input signals. In particular, the reset signal should make the $\Im$ return to an initial state, whereas the inhibit signal should make it go to a specific state in which all output actions are disabled. The subsumption interaction between layers can be modeled by a simple $\Im$, the *subsumption*-$\Im$. This $\Im$ will have as input signals the output of a dominant module and that of an inferior module. Its output signal will be identical to the inferior input signal as long as the dominant input signal is absent.[7] Otherwise, it will be identical to the dominant input. Obviously, the *subsumption*-$\Im$ is just an implementation of a two-level priority scheme. It can be easily extended[8] to model a static multi-level priority scheme[9] which would then represent a hierarchy of dominant and inferior behaviors.

The subsumption architecture is suitable for the specification of task-achieving behaviors that can be *statically* organized as a hierarchy of dominant and inferior behaviors. It cannot deal with applications with *dynamically* changing priorities. In particular, if the priority of a behavior depends on the task (or goal) to be achieved, and if such a goal is dynamically changing, then this behaviour can be dominant in some situations and inferior in others. Rather than dominant and inferior behaviors, such systems are described in terms of *competing behaviors.*

Unlike the subsumption architecture, the $\Im$ framework is ideal for the specification of systems with competing behaviors. To illustrate that, consider the following specification of "Buggy", a cockroach-like robot. Buggy has two actuators that allows it to move in 2-D. The first actuator allows Buggy to either move forward with a constant speed or stop, whereas the second actuator allows it to rotate. Buggy has three sensors: a *proximity sensor*, a *crack locator* and a *food detector*. The proximity sensor provides Buggy with information about predators. In particular, it returns the distance and direction of the closest detected obstacle (wall). The crack locator informs Buggy about any cracks found in its neighborhood. If such a crack is found (within a limited distance), the crack locator returns the vector perpendicular to the crack. The food detector continuously returns a boolean value that informs Buggy about the existence or non-existence of food at its current position.

Buggy has two basically different and often competing behaviors, namely a conscious behavior and a subconscious behavior. Consciously, Buggy searches for food and eats it. Experience (or evolution) has taught Buggy that food is often found along cracks. Thus, to eat, Buggy looks for and moves along cracks. Subconsciously, however, Buggy tries to keep itself away from predators.

For simplicity, we will assume that Buggy has only one goal: to survive.[10] Survival, however, requires both eating (to avoid starvation) and escaping from predators (to avoid being crushed).

[7] Absent can be interpreted in a number of different way. For example it can mean the absence of any actions for more than a given time interval.

[8] by implementing the r-level priority scheme with a binary tree of *subsumption*-$\Im$'s.

[9] Or any other priority scheme.

[10] In a more realistic specification Buggy might have dynamically changing goals.

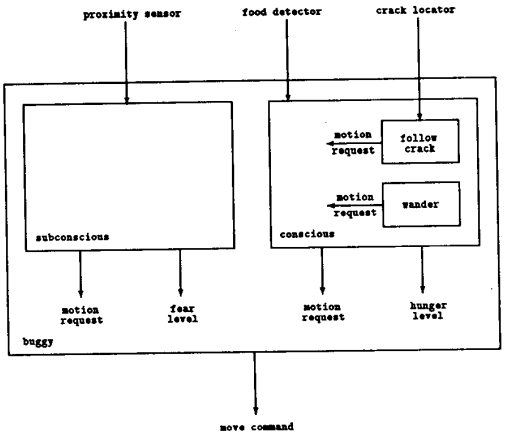



Figure 3: The different $\Im$'s in Buggy's specification

These two behaviors often conflict. Buggy's urge to find food increases as time ellapses and no food is found. On the other hand, Buggy's fear from predators increases as its distance from them decreases. At any point in time, the behavior that is more important to Buggy's survival subsumes the other behavior.

Using the $\Im$ framework, Buggy can be specified as a $\Im$, **buggy**, which itself is described in terms of two $\Im$'s, `conscious` and `subconscious`, corresponding to the conscientious and subconscientious behaviors described above. At any point in time, **buggy** selects one of these two behaviors, based on its urgency level[11] and Buggy's goal at that time.[12] `Conscious`, in turn, is described in terms of two $\Im$'s, **wander** and `follow_crack`, corresponding to the basic behaviors of moving at random and following cracks, respectively. Figure 3 shows the different $\Im$'s and their associated channels.

We have written a full specification of Buggy in ESPRIT.[13] This specification was executed and Buggy's behavior was monitored for different parameters.[14] In figures 4 and 5, we show the outcome of two of these experiments. From these, one can easily recognize the different patterns of wandering around looking for a crack, approaching and following cracks, and avoiding walls. Note the limit cycles caused by the changing hunger and "fear" levels.

The specification of task-achieving behaviors using the $\Im$ model, while more expressive, retains all of the advantages of the subsumption architecture. In particular, it allows us to build up the complete control system incrementally; it allows debugging of individual behaviors in isolation of the others; and, it facilitates the reuse of off-the-shelf basic behaviors.

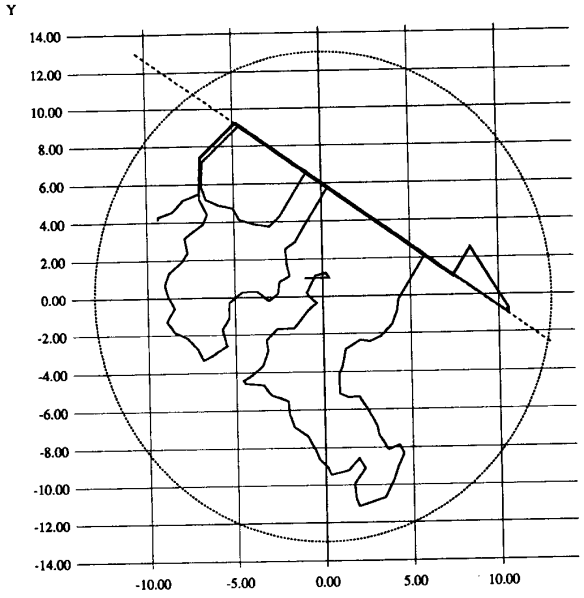
[11] Hunger level for the conscientious behavior and fear from predators for the subconscientious behavior.

[12] In this example Buggy has only one goal: to survive.

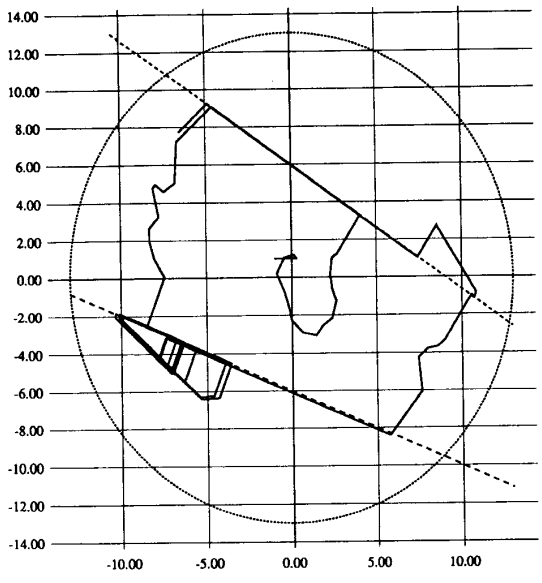
[13] This specification is not included for space limitations.

[14] For example: Buggy's initial conditions, rate of hunger increase/decrease, sensitivity and accuracy of the sensors, location of food along cracks, timing delays and constraints for the different actions, ... *etc.*

**Behavior of Buggy in an Environment with 1 crack**




**Behavior of Buggy in an Environment with 2 cracks**



## 4 Summary

This paper deals with the management of conflicting demands on the use of the motor units of the robot from the active sensors and manipulatory processes.

We introduced an "operating system" like facility for managing the conflicting motor requests produced by multiple active sensing and manipulation tasks. We describe a proposed implementation of this sensori-motor management facility based on the Input-Output-Timed-Automata (IOTA) abstraction. The IOTA abstraction is general enough to allow the enforcement of temporal constraints and to permit the specification and modification of task priorities based on the goals and current state of the robot. The IOTA based sensori-motor operating system acts as a scheduler for various motor requests submitted by the active sensor and manipulation systems. Our proposed sensori-motor system is more general than the subsumption architecture of Brooks [7] and can implement in a straightforward fashion alteration of priorities.

## References


[1] Aloimonos, Y., Weiss, I., Bandyopadhyay, A., "Active vision", Proceedings of the 1st IEEE Conference on Computer Vision, London, 1987, pp. 35-54

[2] Bajcsy, R., "Active perception vs. passive perception", Proceedings 3rd IEEE Workshop on Computer Vision, Bellaire, pp. 55-59, 1985

[3] Bangs, A., "The Motion Interpreter/Control Computer System", Harvard University Undergraduate Thesis, 1988.

[4] Bestavros, A., "A new environment for developing real-time applications based on the IORTA model", *Internal Report - Department of Computer Science, Harvard University*, April 1989.

[5] Bestavros, A., "The input output real-time automaton: A model for real-time parallel computation", Technical Report TR-12-89, Department of Computer Science, Harvard University, 1989.

[6] Brockett, R.W., "On the computer control of movement", Proceedings of the 1988 IEEE Robotics and Automation Conference, Philadelphia

[7] Brooks R., and Connell, J., "Asynchronous Distributed Control System for a Mobile Robot", *SPIE Proceedings*, Vol. 727, October 1986.

[8] Burt, P., "Algorithms and architectures for smart sensing", in the Proceedings of the 1988 Darpa Image Understanding Workshop, pp. 139-153

[9] Clark, J., and Ferrier, N., "Modal Control of an Attentive Vision System", in the Proceedings of ICCV, Florida, December 1988.

[10] Eng, V., **Design and Implementation of a Motion Description Language,** Ph.D. Thesis, Division of Applied Sciences, Harvard University, to appear.

[11] Flynn, A., and Brooks, R., "MIT mobile robots - What's next?", Working paper 302, MIT AI Lab, November 1987

[12] Hoare, C. A. R., "Communicating sequential processes", *Communication of the ACM*, Vol. 21, August 1978.

[13] Stankovic, J.A., "Misconceptions about real-time computing", *IEEE Computer*, October 1988.

[14] Stansfield, S., "Visually aided tactile exploration", Proceedings of the 1987 IEEE Conference on Robotics and Automation, pp 1487-1492