Proceedings of the 1996 IEEE
International Conference on Robotics and Automation
Minneapolis, Minnesota - April 1996

A System for Active Vision Driven Robotics

Jeffrey A. Fayman, Ehud Rivlin

Computer Science Department
Technion, Israel Institute of Technology
Haifa 32000, Israel

Henrik I. Christensen

Laboratory of Image Analysis
Aalborg University
DK−9220 Aalborg East, Denmark

Abstract

In this paper, we present an agent architecture/active vision research tool called the Active Vision Shell (AV-shell). The AV-shell can be viewed as a programming framework for expressing perception and action routines in the context of situated robotics. The AV-shell is a powerful interactive C-shell style interface providing many capabilities important in an agent architecture such as the ability to combine perceptive capabilities of active vision with capabilities provided by other robotic devices, the ability to interact with a wide variety of active vision devices, a set of image routines and the ability to compose the routines into continuously running perception action processes. At the end of the paper, we present an example of AV-shell usage.

1 Introduction

In recent years, active vision has become an intensive area of robotics and vision research. Active vision, which was first introduced in [2] and later explored in [1], is defined as the explicit control of the sensory system to improve robustness and eliminate ill-posed conditions. It has been shown that by actively controlling the vision system, many classic computer vision problems become easier to solve [1].

The integration of active vision into more complex robotic systems requires active vision devices to provide high quality visual information in dynamic real-world situations. To provide this information, active vision adopts the behavior based approach in which a tight coupling between perception and action is exploited to achieve particular goals in continuously running visual systems.

In more complex systems in which the visual information provided by active vision is used to drive other robotic devices, multiple levels of perception and action coupling must exist. For example, the micro-level coupling required to drive the active vision device and macro-level coupling between the active vision device and other robotic devices.

In this paper, we present the Active Vision Shell (AV-shell), an agent architecture/active vision research tool. The AV-shell provides capabilities important in an agent architecture such as the ability to interact with a wide variety of active vision devices, a set of image routines and the ability to compose the routines into continuously running perception action processes.

The remainder of this paper is organized as follows: In section 2, we review related work. In section 3 we discuss the design goals and methods used in the development of the AV-shell. In section 4 we present the methodology adopted in the AV-shell for composing routines into continuously running perception action processes. In section 5 we present the AV-shell, discuss its origins and briefly discuss its interface. In section 6, we discuss a control hierarchy adopted in the AV-shell appropriate for active image processing as well as integrating active vision into more complex robotic systems. In section 7, we present an example of how the AV-shell can be used to implement complex perception action tasks and we conclude with section 8.

2 Related Work

This paper focuses on the the AV-shell and its components for composing and integrating elementary robotic and vision routines into continuously running perception action processes.

Relatively little work has been done in integrating active vision into more complex systems. In [4], Crowley and Christensen discuss a sophisticated architecture and system called "VAP/SAVA" for the integration and control of a real-time active vision system.

Their system provides a comprehensive agent architecture which was designed to accommodate a continuously operating process. The VAP/SAVA system was designed specifically for the purpose of demonstrating continuous operation in the context of an interpretation. It possesses many of the features required in a system for active vision experiments, however it is desirable to have programming facilities that enable flexible composition of processes to enable a variety of architectures and applications. Several other active vision environments are described in [3].

Process composition by representing task/plans as networks of processes was first proposed by Lyons [9] where he discusses the Robot Schema model. Kosecka and Christensen [7] adopt Robot Schemas and show how one can synthesize a finite state machine supervisor which serves as a discrete event controller. Elementary behaviors appropriate in the domain of an "intelligent delivery agent" are described and experiments in robot navigation are presented. Our work also makes use of the Robot Schemas model, however, our elementary behaviors are derived from the active vision domain and we present a system for accessing these behaviors.

3 AV-Shell Goals and Methods

We believe that the full realization of the benefits of active vision will not occur until active vision is studied in broader contexts. In biological systems the visual system and the apparatuses used for its movement perform in a way that can be described as reflexive, providing the perceptive capabilities used in a perception-action cycle. The actions in this cycle are in turn carried out by other parts of the system. This implies that perception and action must be combined to provide capabilities similar to those found in complex systems. It also necessitates powerful and flexible architectures. The AV-shell can be viewed as playing dual roles: (1) as suggesting an agent architecture framework; and (2) as an active vision research tool.

As an agent architecture framework, the AV-shell provides tools necessary for efficient integration of active vision into complex systems. As an active vision research tool, the AV-shell provides an intuitive interface to an extensive set of active vision and robotics related commands, the ability to compose these commands into continuously running perception action processes and a full programming language.

Several goals guided the design of the AV-shell. Firstly, it should be applicable to a wide variety of active vision devices. This helps to make it portable and gives us the ability to experiment with various active vision configurations using the same software. Secondly, it should provide a modular vocabulary of visual routines including high-level activity routines, medium-level basic process routines and low-level primitive routines. It should also provide the architectural flexibility to experiment with various combinations of these visual modules. Thirdly, the AV-shell should provide a convenient platform for performing experiments in these areas. Lastly, it should be based on an architecture which both enables the various components of active vision to be integrated, and allows active vision to be incorporated into more complex systems.

3.1 Active-vision devices

We explained earlier that the primary function of an active vision device is to allow for active movement of the visual sensors thereby making previously ill-posed problems well defined. This implies that the device is simply a manipulator used for moving the visual sensors. Various paradigms such as moveable eyes, eye-in-hand, robotic heads and heads-in-hand have been used to do this.

It is desirable to provide an abstraction for a general interface to this wide range of devices. At the same time, we would like encapsulation to enable transparent use of existing control software. This naturally leads to an object oriented approach and because we conceptually have difficulties supporting both perception and action in one device, we believe that it is necessary to view the vision sensors and the mechanisms used to move them separately. We define an "Active-vision device" (AVD) to include two parts: the vision sensor, which we refer to as "camera" and the device used for moving the camera, which we refer to as "head". By doing this, several benefits are realized. Firstly, the mechanism will be applicable to stationary vision systems as well; these are simply cameras without the ability to move. Secondly, by decoupling cameras and heads, the "vision" problems and the "active" problems of active vision can be studied individually or together.

Another benefit we gain by this view of AVD's is that we can define their operational space in terms of the cameras alone without concern for the underlying movement mechanism. This leads to a set of medium-level commands that are compatible with many active vision devices.

3.2 Active vision routines

Before we discuss the composition and integration facilities of the AV-shell, we will briefly mention the structure of the active vision routines provided by the AV-shell. A thorough discussion of these routines can be found in [5].

The active vision routines provided by the AV-shell are made up of three "levels"; (1) high-level activity routines supporting activities such as fixation and pursuit; (2) medium-level basic process routines such as accommodation and motion detection; and (3) low-level primitive routines such as convolution and correlation. In this configuration, processes at higher levels are made up of processes at lower levels.

4 AV-shell Process Composition

While the routines mentioned previously constitute a rich set of active vision routines, the task of composing them into continuously running perception action processes has not been specified. In this work, we adopt a model proposed in [9] called the Robot Schemas (*RS*) model. Table 1 summarizes the *RS* composition operators. [1] In the *RS* model, communication channels between concurrent processes are called "ports". Messages are written to, and read from ports. A port to port *connection relation* can be specified as an optional third parameter in concurrent composition. This connection relation specifies a set of couples $op \mapsto ip$ indicating that port ip and op are connected.

Using Robot Schemas notation, we see how the temporal and structural dependencies required to implement the high-level activities of fixation and pursuit are captured (stabilization is similar to pursuit). Fixation is initiated with a saccade to the fixation point followed by continuous vergence control driven by disparity and accommodation cues.

$$\text{fixation} = \text{saccade} : ((\text{disparity} \mid \text{accommodation}) \;{:};\; (\text{vergence control}))$$

In pursuit, vergence, foveal motion detection (FMD) and dynamic accommodation are used to continuously drive motion of the vision sensor.

$$\text{pursuit} = (\text{vergence} \,\#\, \text{FMD} \,\#\, \text{dynamic accommodation}) \;{:};\; \text{move}.$$

[1]Notation in table 1 is consistent with those of [9], however, in our implementation, the operators were changed to avoid conflicts with existing operators.

1. *Sequential Composition*:
   $T = P; Q$. The process T behaves like the process P until that terminates, and then behaves like the process Q (regardless of P's termination status).
2. *Concurrent Composition*:
   $T = (P|Q)^c$. The process T behaves like P and Q running in parallel and with the input ports of one connected to the output ports of the other as indicated by the port-to-port connection map c. This can also be written as $t = (|_{i \in I} P_i)^c$ for a set of processes indexed by I.
3. *Conditional Composition*:
   $T = P\langle v \rangle : Q_v$. The process T behaves like the process P until that terminates. If P aborts, then T aborts. If P terminates normally, then the value v calculated by P is used to initialize the process Q, and T then behaves like Q_v.
4. *Disabling Composition*:
   $T = P\#Q$. The process T behaves like the concurrent composition of P and Q until either terminates, then the other is aborted and T terminates. At most one process can stop; the remainder are aborted.
5. *Synchronous Recurrent Composition*:
   $T = P\langle v \rangle :; Q_v$. This is recursively defined as $P :; Q = P : (Q; P :; Q)$.
6. *Asynchronous Recurrent Composition*:
   $T = P\langle v \rangle :: Q_v$. This is recursively defined as $P :: Q = P : (Q|(P :: Q))$.

Table 1: Summary of *RS* Composition Operators

Each active vision routine has clearly defined initiation, terminations and interruption mechanisms. Therefore, the routines can be represented as finite state machines. Using this representation, system behaviors may be formally analyzed. Composition of active vision routines is achieved by traversing a parse tree built from an expression in this notation.

An example of such a parse tree is given for fixation in figure 1.

The previous discussion of process composition applies to robotic processes as well. For example, a task which integrates an active vision device with a manipulator for repeatedly locating and grasping parts from a moving conveyor can be specified as follows:

$$\text{capture} = (\text{find part} : \text{fixate}) \;{:};\; ((\text{pursuit} \mid \text{locate part}) : \text{grasp part})$$

In the example, we first find the part on the conveyor, then fixate on it. Once fixated, we pursue it with the active vision device which provides data to

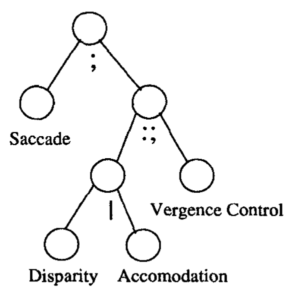

Figure 1: Fixation Parse Tree

the arm for locating to the part. Finally we grasp the object. The purpose of the example is to illustrate the expressive power of the composition operators rather than how each primitive process is implemented.

5 The AV-Shell

In this section we present the AV-shell. The active vision components of the system provide the ability to control a wide variety active vision devices while integrating active vision into complex robotic systems is accomplished by combining the AV-shell with another system called the Robot Shell (R-shell) [11, 10, 6]. The R-shell provides the tools necessary for controlling and coordinating various robotic devices such as manipulators and dextrous robotic hands. However, it does not possess active vision capabilities or the process composition capabilities required in behavior based systems. We begin with a brief overview of the R-shell, then discuss the AV-shell.

5.1 Robot Shell

The R-shell is an interactive program written in C under the UNIX operating system. It interprets and executes robotics related commands called "R-shell commands" in much the same way that a shell such as c-shell interprets and executes system related commands. Robotic data in the R-shell environment are entered, manipulated and displayed through the use of these commands. Included in the set of R-shell commands are an extensive set of algebraic operations such as matrix and vector addition, multiplication, matrix inversion and transposition, computation of determinants, traces and norms as well as various arithmetic operations and standard mathematic functions for scalars all of which are used extensively in robotics work.

R-shell commands can be entered interactively or placed in R-shell scripts and executed by simply typing the script file name followed by its actual parameters. Thus scripts provide procedural abstraction in R-shell. Scripts can call other scripts and can call themselves providing nesting and recursion. The R-shell script language provides a full set of flow control statements such as if-then-else, while, loop and case. Additionally, R-shell supports both call-by-name and call-by-value methods for passing parameters.

5.2 AV-shell

The AV-shell provides capabilities missing in R-shell for handling active vision devices and behavioral composition. The interface consists of four groups of commands: camera, head, active image routine and general. Camera commands are used to control cameras and camera lenses, head commands are used to control heads, active image routines provide the levels of active image functions provided by the AV-shell and general commands are R-shell commands which have been extended to handle active vision devices. A thorough review of important AV-shell and R-shell commands can be found in [5].

The following are examples of a camera command and an active vision routine command respectively.

s2c
: Map a screen space coordinate to a Cartesian space coordinate using the calibration matrix in the standard AV-shell variable **C**.

fixate $[x, y, z]$
: Cause the optical axis of the cameras attached to head *hsel* to intersect at the 3D point given in x, y, z.

6 Combining Perception and Action

Combining the perceptive capabilities of AVD's with the action capabilities of other robotic devices in an agent architecture poses some interesting problems. On one hand, we must view perception and action on a micro-level looking only at the AVD as it operates independently in an perception-action loop. On the other hand, at the macro-level, the perceptions provided by the AVD are used to drive actions of other robotic devices. In this section we will look at architectural issues related to the combination of perception and action at both the micro and macro levels.

The control of robotic devices can be conveniently divided into three levels: **high**, **medium** and **low**. Figure 2 shows this relationship.

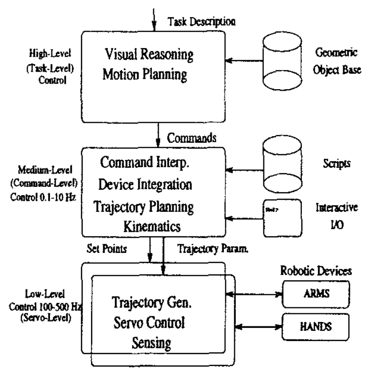

Figure 2: Typical Control Hierarchy

High-level control usually refers to task-level planning. At this level, reasoning is incorporated into the control system with methods such as knowledge-based systems, fuzzy logic and artificial neural networks. An example is the reasoning behind chess playing algorithms. Typical output from the high-level includes Cartesian information such as the position and orientation of various pieces on the chess board and to where they should be moved.

The medium-level takes output from the high-level and converts it into a form that is suitable to the joint controllers. In the case of the chess playing robot, the medium-level will convert the Cartesian position and orientation information of the pieces provided by the high-level into the joint values needed by the robot at the low-level to move the pieces. As an aid to research, the medium-level should provide an interactive interface so that researchers can bypass the high-level task planners.

Low-level controllers deal with trajectory generation, sensor integration and servo joint control. This is the level where connection to physical devices is made.

6.1 Active-Vision integration and Active-Image processing

While this hierarchy is appropriate for devices such as manipulators, it is insufficient for active image processing and the various compositions of active vision routines, both of which are required in AVD's. Active vision processing is needed to compose lower level active image routines into higher level activities. These higher level activities in turn must be composed according to changing tasks and and effectively invoked. Effective invocation requires system delays to be compensated for by the controller (using a Smith-controller or other techniques).

In order to modularize these various levels of processing, we feel that in the case of AVD's, two additional levels are needed in the control hierarchy of figure 2. The new levels are called the "Active Vision Integration Level" (AVIL) and the "Active Image Processing Level" (AIPL). The hierarchy of figure 2 has been augmented with the new levels. The augmented hierarchy is shown in figure 3.

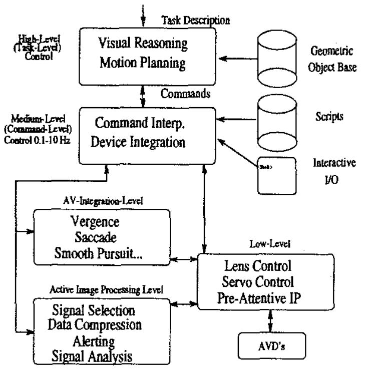

Figure 3: Augmented Control Hierarchy

According to the diagram, the low-level is responsible for providing pre-attentive image processes. The AIPL performs filtering and and fusion to provide data to the set of attentive routines found in the AVIL. The AVIL in turn is responsible for composing the higher level primitives according to changing requirements and compensating system delays. All three of these levels are accessible from the command interface at the medium-level facilitating development and simulation.

7 Example AV-shell Usage

In this section, we will demonstrate how the AV-shell can be used to easily implement a complex perception-action task. In the example, the AV-shell will be used to coordinate the activities of three robotic devices: a manipulator, a dextrous robot hand and an AVD (head). The scenario, which is illustrated in figures 4 and 5(a,b), consists of an AdeptOne manipulator with an attached Belgrade/USC dextrous hand [6]. Additionally, there is a table in the workspace of the AdeptOne on which a cylindrical toy train is moving. Observing the movement of the train is an TRH robotic head [8]. The goal of the example is to have the head track the train and signal when the train is about to fall off an edge of the table. The arm waits for the signal at which point it is commanded to bring the hand to the train so that it can be grasped before it falls.

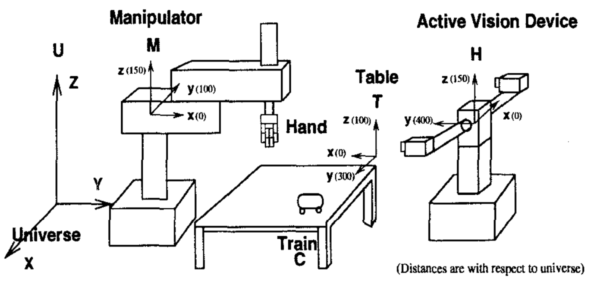

Figure 4: Example Environment

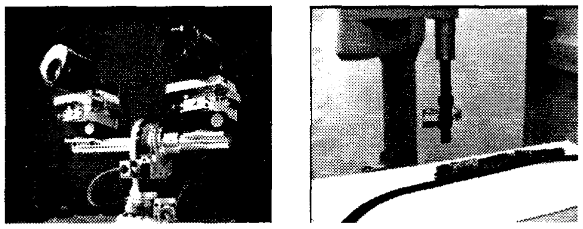
(a) (b)

Figure 5: (a) TRH Head; (b) AdeptOne and Train

The AV-shell script necessary to perform this task is made up of two parts. In the first part, the internal representations of the environment and devices are created. This includes creating and locating the manipulator, hand, table, cameras and head. In the second part, we activate the head and perform the task.

Creating the internal representation of the environment in the AV-shell includes defining parameters of the various components present. Each robotic device has a set of parameters required for its control which must be defined. This includes things such as minimum and maximum joint values etc... Additionally, the components must be initially located relative to the universe and devices attached to other devices (i.e. hand on manipulator and cameras on head) in reality must also be attached in the internal representation. Also, the description of the train must be memorized by the AV-shell.

Once the environment has been defined, we are ready to begin the task. The first step is to connect the internal description of the devices to their physical counterparts. This is accomplished with the connect command. Once connected, commands causing changes in the internal configuration will effect the physical devices as well. If we execute the task without connection to the physical devices, we have simulation. Next, using the AV-shell commands *find* and *saccade*, we locate the train in the environment, invoke a saccade to its position and begin smooth pursuit tracking (this is an example of where perception-action takes place at the micro-level of the active vision device). When the train approaches an edge of the table, the event e1 is set signaling the arm to bring the hand to the estimated falling point of the train (this is an example of where perception-action takes place at the macro-level of the integrated system). At the same time, the hand is preshaped for grasping the train. When the arm has brought the hand to the grasping point and the hand is preshaped, the train is grasped.

In this example, specific robotic devices were used. The same code is applicable to any other devices known to the AV-shell. For instance, we could have used a PUMA manipulator and a KTH head instead. The AV-shell script for performing this task is given below.

```
##########################################################################
#                   Create environment for grasp task                   #
##########################################################################

#------------------------------------------------------------------------#
#Create and locate manipulator:                                          #
#Here, we define parameters for the adeptone manipulator type, create an #
#instance called M and locate it relative to the universe.               #
#------------------------------------------------------------------------#
cop -a                  # default category option is -a (arm)           #
type adeptone           # create arm type name                          #
par a 325.0 375.0 0.0   # define link parameters                        #
    .......

inst adeptone M         # create an arm instance called M               #
loc                     # locate arm M at Universe                      #
tran 0 100 150          # translate arm base                            #
rot -z 90               # rotate arm base about z-axis by 90 degrees    #

#------------------------------------------------------------------------#
#Create and locate dextrous hand:                                        #
#Here we define parameters for the USCBGD hand type, create an instance  #
#called H and attach H to M.                                             #
#------------------------------------------------------------------------#
cop -h                  # change default category option to -h (hand)  #
type USCBGD             # create hand type name                         #
par gmin -20, -45, -90, -90 # define hand parameters                    #

    .......

inst USCBGD H           # create a hand instance called H               #
open M                  # prepare manipulator for attachment            #
attach H M              # mount hand onto arm                           #
close M                 # disable further attachments                   #

#------------------------------------------------------------------------#
#Create and locate table:                                                #
#Here we create the table and locate it relative to the universe         #
#------------------------------------------------------------------------#
cop -o                  # change default category option to -o (object) #
obj T                   # create table name                             #
loc                     # locate table at Universe                      #
tran 0 300 100          # translate table base                          #
rot -z -90              # rotate table base about z-axis by -90 degrees #

#------------------------------------------------------------------------#
#Create and locate head:                                                 #
#Here we define parameters for the TRH head type, create an instance     #
#called HD and locate it relateive to the universe.                      #
#------------------------------------------------------------------------#
cop -d                  # change default category option to -d (head)   #
type TRH                # create head type name                         #

    .......
```

```
inst TRH HD                  # create a head instance called HD           #
loc                          # locate head HD at Universe                 #
tran 0 400 150               # translate head base                        #
rot -z 180                   # rotate head base about z-axis by 180 degrees #

#--------------------------------------------------------------------------#
#Create and locate cameras:                                                #
#Here, two cameras CL and CR are created and attached to HD.               #
#--------------------------------------------------------------------------#
cop -c                       # change default category option to -c (camera) #
type cohu                    # create camera type name                    #
inst cohu CL CR              # create camera instances called CL and CR   #
open N                       # Prepare head for camera attachment         #
sel CL                       # select camera CL                           #
attach CL HD                 # Mount CL onto HD                           #
trans -o -x 50               # translate camera CL to final position on HD #
sel CR                       # select camera CR                           #
attach CR HD                 # Mount CR onto HD                           #
trans -o -x -50              # translate camera LR to final position on HD #
close HD                     # disable further attachments on HD          #

#--------------------------------------------------------------------------#
#Memorize description of the train                                         #
#Here, the train is presented to the AV-shell which memorizes its description#
#--------------------------------------------------------------------------#
memorize train                 # AV-shell memorizes the image of the train #
select train                   # make car the selected object             #

############################################################################
#                       Tracking and grasping the train                    #
############################################################################
event e1 e2                  # create event names                         #
connect H  1                 # connect manipulator H with real arm (port 1) #
connect HD 2                 # connect head HD with real head (port 2)    #
connect N  3                 # connect hand N with real hand (port 3)     #
cop -d                       # default category option is -d (head)       #
find train                   # locate the train in the environment        #
saccade ; pursuit -h HD:e1   # saccade to the train and begin tracking    #
                             # set e1 when train approaches table edge    #
                             # This includes estimate of falling location #
wait e1                      # wait for event e1 to occur                 #
j2g                          # Train relative to cyclopean frame of HD.   #
where -o                     # transform to world coordinates             #
cop -h                       # default category option is -h (hand)       #
loc train:e1                 # move hand to grasp frame then set event e1 #
shape -g7 70:e2              # preshape hand for cylindrical grasp with 70% #
                             # aperture.  Set event e2 when done          #
wand e1 e2                   # wait for both events to occur              #
grasp 2.5                    # grasp the train with force of 2.5 N        #
wait                         # wait for grasping to complete              #
trans -h -z 90               # lift the train                             #
```

8 Conclusion

In this paper, we addressed issues related to agent architectures in active vision research. We presented the agent architecture/active vision research tool called the AV-shell which provides a programming framework for expressing perception and action routines as well as the techniques necessary to integrate a wide range of active vision devices into complex robotic architectures. The AV-shell is built up of a set of active vision routines along with the ability to compose these routines into continuously running perception action tasks. Future work on the AV-shell includes research into providing fault tolerance.

References

[1] Y. Aloimonos, I. Weiss, and A. Bandyopadhyay. Active vision. In *International Journal on Computer Vision*, pages 333–356, 1987.

[2] R. Bajcsy. Active perception vs passive perception. In *Proceedings of the Third IEEE Workshop on Computer Vision*, pages 55–59, Bellaire, Michigan, 1985.

[3] H.I. Christensen and J.L. Crowley. *Experimental Environments for Computer Vision and Image Analysis.* World Scientific Press, 1994.

[4] J.L. Crowley and H.I. Christensen. *Vision as Process.* Springer-Verlag, January 1995.

[5] J.A. Fayman, E. Rivlin, and H.I. Christensen. The active vision shell. Technical Report CIS Report 9510, Israel Institute of Technology – Technion, May 1995.

[6] Jeffrey A. Fayman. Medium-level control of robot hands. Master's thesis, San Diego State University, Department of Mathematical Sciences, San Diego, CA, 1990.

[7] Jana Kosecka and Henrik I. Christensen. Experiments in behavior composition. In *Proceedings of the 3rd International Symposium on Intelligent Robotic Systems*, pages 129–139, Pisa, Italy, July 1995.

[8] O. Kraft and J.A. Fayman. Trh - low level hardware and software. Technical Report RLR Report RLR0003, Israel Institute of Technology – Technion, April 1995.

[9] Damian M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, 9(7):241–256, June 1993.

[10] M.I. Vuskovic. R-shell: A unix-based development environment for robotics. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Philadelphia, Pennsylvania, April 1988.

[11] M.I. Vuskovic, A.L. Riedel, and C.Q. Do. The robot shell. *International Journal of Robotics and Automation*, 3(3):165–175, 1988.