Proceedings of the AAAI Spring Symposium, 1995.

A Constraint-Based Control Architecture for Acting and Reasoning in Autonomous Robots

Justinian P. Rosca and Terry Riopka
Computer Science Department
University of Rochester
Rochester NY 14627
rosca@cs.rochester.edu, riopka@cs.rochester.edu

Abstract

In this paper we will address several architectural decisions in defining a software control architecture for mobile robots. Our system is a collection of control primitives that enables the development of simulations or control algorithms for autonomous agents. Its computational capabilities are determined by an object-oriented constraint-based architecture. We discuss how high level knowledge, skills, goal-driven and reactive behavior are integrated within such an architecture. Our goal is to design a framework that enables the merging of classic and reactive implementation ideas. We will show, that each such type of control can be implemented in our system. The issues of task decomposition and granularity are given special attention, as they lie at the basis of our architecture. We discuss two learning methods supported by our system. The first is based on environment exploration, while the second copes with skill acquisition. Our robot, CYCLOPS , is a LEGO mini-robot based on the 6.270 MIT kit to be used in a package delivery application.

Introduction

Hardware miniaturization offers the opportunity to bridge the gap between laboratory mobile-robot designs and industrial applications like floor cleaning, surveillance, flexible part transportation, delivery, or harvesting [13].

Each of the applications in this extended context requires increased sensing capabilities and processing power. System architectures are beginning to reflect the increase in complexity by integrating multiple micro-controllers and relying on distributed processing. Consequently, software architectures have to cope with an increased range of complex problems and need to answer questions such as how global functionality can be achieved, how knowledge and skills are acquired and robots are programmed and tested, how robots can learn to improve their task performance, how performance degrades in case of module failures and how the system copes with uncertainty and imprecision, to name just a few.

The two prevalent architectural ideas are the classic or centralized approach and the reactive approach ([1], [2]). They differ in the way each views the role played by a world model. The centralized or knowledge-based approach emphasizes a (generally declarative) model of the world by representing detailed expertise from the problem domain [11]. It integrates sensing, reasoning or problem-solving and acting modules [15] that operate on the domain knowledge. The reactive or behavior-based approach recognizes the importance of the principles of connectedness and embodiment (the robots experience the world directly). Consequently, the system must get beyond symbolic reasoning to consider special purpose reflexes that enable the robot to act at high speeds in certain circumstances, to be "situated" in its environment. No central world model is used in this latter approach, as the world can be directly sensed depending on the agent's goal and situation.

In this paper we will address several architectural decisions for building artificial agents by taking a hybrid approach. We aim at embodying these principles in a collection of primitives that will enable an easier prototyping of applications. We present the main features of our collection of primitives called the software architecture control kit (ACK). We discuss two learning methods we are interested in. The first is based on environment exploration, while the second copes with skill acquisition. Finally we present our robot, a LEGO mini-robot based on the 6.270 MIT kit and a delivery application which is currently under development.

Design Decisions

The computational capabilities of our system will be determined by an object-oriented architecture. We discuss how high level knowledge, skills, goal-driven and reactive behavior can be integrated in a constraint-based architecture.

The major design goals we want to address are:

- facilitate a distributed and modular software organization
- enable the implementation of both goal-driven activities and reactive behaviors.

- sense time and take into account time constraints, both for guiding the system actions and for improving its behavior.
- enable active learning by deciding what to be explored and by adapting the robot to the environment in which it operates
- enable evolution towards more complex or more refined patterns of operation

Although our test robot hardware does not allow the connection of additional extensions based on microcontrollers, we consider that the software architecture should aim at a multi-controller network as in [13]. Moreover it should make such an architecture transparent to the programmer. A modular architecture organizes the knowledge, competences and activity of the system such that local changes, improvements or tuning do not affect the functioning of unchanged components. It offers a higher level of abstraction that can be coped with in an easier way.

Goal-driven behavior is an important characteristic of an intelligent system. It influences the types of control mechanisms and knowledge representation we adopt. Goal-directed activities ensure global system functionality. Traditionally, symbolic reasoning techniques have been successfully applied for implementing goal-driven planning and problem solving. Real-world situations do not always allow spending time for symbolic inference. Reactive behaviors representing independent competence modules can both sense the world and control the system actuators in order to respond fast to the current world situation. However, such an approach does not scale well. As the number of modules implementing behaviors gets larger, their possible interactions become significant and it becomes harder to ensure that modules do not work at conflicting goals. Integrating higher level reasoning processes and reactive behaviors may inherit the advantages of both approaches while solving some of the problems each separate approach has.

The situatedness property implies that the control system is able to react in a timely fashion to various stimuli. Time allows the formulation of simple plans, meta-reasoning or simple reinforcement of various actions taken during goal pursuance. Time represents an essential element in the integration of reactive behaviors with high-level reasoning and learning.

The Architecture Control Kit

The level of parallelism and modularity of a distributed system depends on conceptual factors such as: program granularity, parallelism control and process synchronization and communication.

Parallelizable numerical applications are characterized by granules loosely coupled from the point of view of their semantics. A granule is a group of program units that can be executed in parallel with other such groups. Robotics and knowledge processing techniques address applications with much higher run time dynamics. The detection of granules is highly dependent on the environment and on the problem solving context ([8], [9]). For example, behavior selection in an artificial creature depends on the current situation given by sensor data, by the creature's motivations and beliefs [10]. The proposed ACK will try to exploit this dynamic type of context-dependent parallelism that results from the interaction of problem entities.

Knowledge representation

A symbolic, object-oriented representation of the application domain, although not exhaustive, makes it possible to name and control the chunks of work the system has to do at a given time.

The knowledge representation paradigm used is inspired by the blackboard model [4] and an implementation of it that outlines the tradeoff between granularity and parallelism in knowledge processing [3].

The main static representational entities are the *object* and the *knowledge source*. Knowledge sources can be activated to generate *knowledge source activation records* (KSAR) which are used to record and control the dynamics of the system.

Objects are described using a pattern called *object prototype*. We explicitly talk about instances of an object as objects created dynamically, according to a given prototype. An object prototype consists of descriptions of object attributes, knowledge sources attached to the object and connections to other objects attached to object attributes. Attributes have symbolic names and carry a simple or a structured value. Values are implemented by means of simple C types.

Knowledge sources are the second important type of system entity. They specify the way in which objects are processed. Any fragment of code, no matter how simple or complicated, can be encapsulated in a knowledge source.

An object's connections to other objects are interpreted as object relations between the object and other system objects. Relations are themselves implemented by system objects, carrying knowledge sources that may specify constraints between objects, how objects influence one another, hierarchical relations between objects, etc.

The system may represent each input sensor data that is individually addressable as an object whose attributes are shared (can be read or written) by all system processes. A particularly important attribute is the sensor reading. Other objects represent system beliefs and are part of a fixed symbolic representation of the problem domain. However, the attributes of such objects are not necessarily shared by all the system processes.

Knowledge sources attached to object relations may be designed to work similarly to rules in a rule-based system, namely they are defined in terms of two condition parts, called activation and execution shields in

[3] and a body.

$$WHEN < activation\_shield(KS) >$$

$$IF < execution\_shield(KS) >$$

$$THEN < body(KS) >$$

The activation shield tests the opportunity or suitability to execute the main part of the knowledge source, the KS body. The execution shield tests the appropriateness to execute the KS body.

These knowledge source components are called fragments. As opposed to a rule-based implementation, KS fragments are implemented as code in the basic implementation language (C in our case). We represent skills as knowledge source fragments.

The main automatic processing mechanism of the system is knowledge source activation. Whenever the value of the attribute of an object changes, all knowledge sources of the object relations attached to the attribute are activated. We will justify how such a simple processing mechanism can generate useful control behavior.

Control architecture

Knowledge processing in such a system is opportunistic ([4]). When knowledge sources attached to objects are activated they create high level, virtual inference processes represented in ACK by knowledge source activation records (KSAR). Executions of KSARs result in computation of new values for the attributes of other objects, activating other knowledge sources and henceforth generating new high level inference processes. Granules of computation are naturally represented by KSARs. If two different KSARs access different objects, to read or write values of attributes for different objects, they are independent and can be executed in parallel. If they access the same object they are serialized in accessing the object. The two KSARs will be executed in a fixed order, according to their priority.

By enabling the description of granule computations, the system makes it possible to parallelize the application. Moreover the way this is done is most general, and can be equally applied to any application built on top of system.

A typical robotics application may have lots of high level processes which correspond to knowledge source fragments activated to solve a problem by coordinating their behavior or working opportunistically either against one another or cooperating to solve the problem. Ideally, inference processes should be independent of one another, so that they can be conceptually run on a parallel virtual machine. ACK provides to the user a model of parallelism at this high level [14].

In the following sections we present two simple examples showing how reactive operation as well as goal-driven operation can be modeled using the system.

Reactive operation We show that one can implement a subsumption architecture in our system by giving a simple example involving a definition of two behaviors and an implementation of a suppressor (or inhibitor) node.

Consider two behaviors, LINE-FOLLOW and AVOID-OBSTACLE that represent two different control layers that run in parallel whenever the appropriate sensors fire. The two behaviors may be related by a suppressor node in a subsumption-like architecture [6]. For example, the second behavior subsumes the function of the first in order to generate the effect of a higher level competence.

An example of a prototype and two knowledge sources that implement the line following behavior in ACK is presented in figure 1. We define two object prototypes *line-follow-proto* having three attributes. The first two are real-valued attributes: *error*, representing an estimation of how much the vehicle bypassed the line to the left or to the right, and *steering-command*, representing the computed steering angle. The third attribute, *line-alarm*, is a boolean-valued attribute that is flipped whenever the line detection sensors detect that the vehicle straddles a line. The knowledge source *line-follow-ks* is attached to the line-alarm attribute, while *steer-ks* is attached to the steering-command attribute. This implies that whenever the sensor values indirectly trigger (by means of other KSs) the values of the alarm, a KSAR with the goal of competing for a motor command that corresponds to line following is created. Similarly, when *steering-command* is set, a *steering-ksar* is created. The *line-follow-ks* knowledge source has three fragments:

1. *line-follow-goal-p* tests if this behavior is appropriate. If we want to execute line following whenever a line is found, then this predicate returns a true value always, otherwise a more complex test can be implemented.
2. *line-positioning-error-p* tests if there exists a positioning error with respect to the line followed. If the answer is true, this predicate has a side effect. It modifies the value of the error attribute of the line-follow-obj object.
3. *line-follow-action* implements a steering proportional to the error computed, in case such an error appeared; it sets the value of the steering-command (steering angle) attribute, in order to trigger the activation of a steer-ksar which will eventually compete with similar KSARs created by other processes.

Suppose that both LINE-FOLLOW-KS and AVOID-KS have generated steering KSARs which will compete for an actuator command. They correspond to different steering instances, so each will refer to a different steering angle and will have a different priority. The one with the highest priority will be selected for execution first. The steer KS fragment will include a command to kill all other existing steering KSARs, if any, after

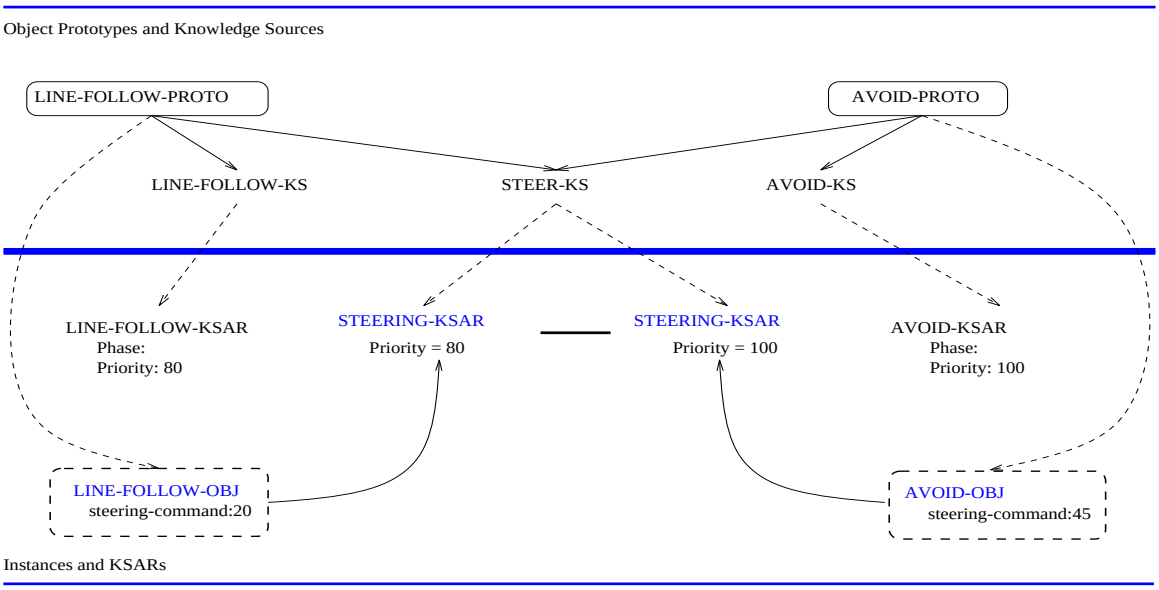

Figure 2: Representation and activation of simple reactive behaviors for line following and object avoidance. The AVOID behavior subsumes the LINE-FOLLOW behavior.

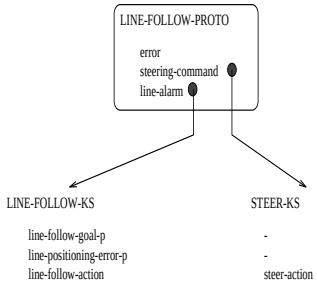

Figure 1: ACK implementation of the line following behavior

steering the vehicle (see figure 2).

Goal-driven activity In the example given above, line-following KSARs are created whenever the left or right reflectance sensors detect a value exceeding a given threshold. Basically, three types of object prototypes were defined, corresponding to sensors, behaviors and actuators. One could similarly define prototypes corresponding to a possible system goal. One of its attributes plays the role of an alarm attribute. Whenever its value is changed by a KSAR or by the main program that starts the ACK scheduler, a process that will work for fulfilling the goal is created. For example, suppose that a neighboring beacon emits an infrared signal on a given frequency. Whenever a signal is detected, the KSAR which continuously monitors such IR signals will modify an alarm attribute. This will be sufficient to trigger the KSARs of a higher level behavior that implement a sequence of actions such as: go to a loading dock, pick up a delivery object, go to the destination and drop the object there.

Reasoning

One interesting characteristic of our architecture is that it can equally implement higher level inference/reasoning tasks as well as low level tasks such as purely reactive, data driven actions. One can combine descriptions of the two types of objects and knowledge sources to implement complicated control systems.

One difference between the two types of processes is that the former type may implement time consuming asynchronous computations, while the latter type usually needs a small amount of time (a system quantum) to execute and may be tightly coupled.

Building an application In order to build up an application, the user has to define the appropriate KB structure (object prototypes, knowledge source prototype, attribute descriptions), the code implementing the knowledge sources and a main program that will start the ACK priority-based scheduler.

Learning

Environment exploration

Learning enables a system to adapt to the environment in which it operates. Adaptive control architectures employ various learning techniques. For example Reinforcement Learning (RL) has been used for learning an (optimal) policy function, mapping situations into actions [16]. If the search process controlled by a RL system chooses the action prescribed by its current

knowledge, then the system *exploits* its current knowledge by acting in order to gain reward. Nonetheless, the system is not endowed with an optimal policy, so that occasionally it should explore actions at random, in order to experiment with more state-action pairs. In this case the system acts to gain information or experience, that is *explores* the search space. The choice of what to do next exemplifies a well-known problem, the exploration-exploitation trade-off (see [5]).

Our approach is to use ACK to describe explicit exploratory behaviors. An example of such a goal is map building. A better estimation of surrounding object positions can affect the execution of system knowledge sources, resulting in improved overall behavior. Map building is one of the background goals in the delivery application. Whenever the robot moves it generates low priority processes for the exploration of neighboring object positions. When no other higher priority processes are active, an arbitration process promotes the execution of the process that attempts to explore the closest area or the one that is supposed to take the least amount of time. The exploratory behavior initiates skills such as "turn left" or "turn right" to precisely examine an area of interest. The exact nature of the exploratory behavior is determined by a combination of sensor information (e.g., the current location of the beacon) and memory (e.g. the expected location of known objects). The result of the exploration is then used to update the map.

Our approach is similar in its character but different in implementation to [12]. [12] describes a control architecture for a mobile robot which combines a reactive subsystem with a search-based planner. A stimulus response subsystem acts when it can, consulting a set of stimulus responses in order to invoke an action in response to current sensed inputs. If no rules apply, a planner is consulted to determine an appropriate action. The results of the planner are then used to acquire a new stimulus response rule, to be activated under the same circumstances in the future.

Skill acquisition

The incremental development of the stimulus response rules mentioned above might be a very difficult task. For a faster and more efficient development, practical robotic systems should be aided to incorporate certain stimulus response rules or "micro-behaviors" by implicit programming through interactive training. This would represent a form of skill rote-learning or learning by being taught.

Skills are sequences of low-level commands that can be automatically applied to generate task-achieving behaviors. We are developing a method for teaching the robot "micro-behaviors". A joystick interface is used to input the necessary motor behaviors and sensor measurements that are associated with a particular set of activation sensor inputs. The system is prompted when to begin recording the micro-behavior and when to stop. For example, wall following could be taught instead of programmed by moving the robot in such a way that it bumps a desired sensor and then move the robot in response to the sensor output. Just relevant sensor data is stored during interactive training. Relevance is defined as what is different.

Robot Description

CYCLOPS is a LEGO mobile mini-robot that is capable of exploring a planar world by combining several high-level behaviors such as line following, beacon following, horizon scanning, obstacle detection and avoidance, wall following and map construction. CYCLOPS can be taught a set of skills which can be used in writing its higher-level behaviors. CYCLOPS has the capability of building a map of a grid-world. Such a map can be used as a starting basis in implementing a simple task planner for optimizing the robot's task in a delivery application.

Our robot (see figure 3) has a variety of sensor capabilities, including infra-red reflectance sensors, modulated infra-red sensor/beacons, an imaging scanner consisting of a single photoresistor in a servo-controlled tube (which we plan to replace with a digital camera on a single chip in the near future), battery power sensor, beam-splitter encoders, and numerous tactile sensors (switches and potentiometers).

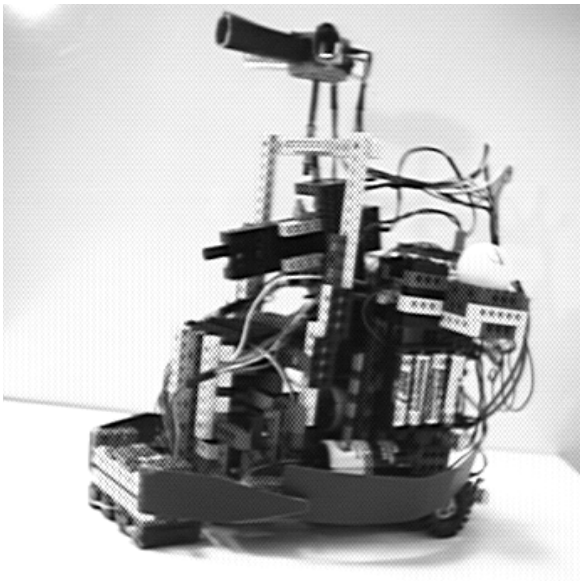

Figure 3: CYCLOPS , the Delivery LEGO Robot

The robot has an 81:1 gear reduced rear wheel drive and a servo-controlled front two-wheel steering. Three rechargeable 5 volt batteries power the driving motor using a separate circuit. Four AA batteries supply power to the logic, all remaining sensors, and two servo motors.

A 32 character LCD display, on-board buttons and a drive/steering joystick provide a primitive interface to the robot vehicle.

Software implementation

Although our system embodies some of the concepts above we are limited by a number of factors, among which the 16K on-board RAM memory and the limited processing power of the MC68HC11 microprocessor board are the most important.

The memory limitation dictates a very simple implementation. We take advantage of the multi-tasking facilities of Interactive C (IC) interpreter, available as free-ware from MIT [6]. KSARs are implemented as processes in IC. The global ACK scheduler is not needed here because the processes created are run using a round-robin multitasking strategy. Each process gets a default maximum number of ticks. The version of IC used does not support structures, so that implementing objects and KSARs becomes awkward. The testing done with the robot aimed at creating all the components for implementing a subsumption-like architecture according to a simplified version of the framework presented. We expect our approach to fully work for a system that supports the ANSI compatible C compiler.

Application

We considered a delivery application. CYCLOPS moves in a world with markings, obstacles and populated by other moving objects. Delivery is requested asynchronously and signaled to the robot using an infrared beacon that emits on a known frequency. Whenever such a signal is issued, if the robot has no other more important goal to pursue it heads towards a loading zone, loads the objects it has to deliver and tries to find its way towards the destination as fast as possible. Delivery and loading zones have fixed locations within a rectangular area. In an extension of the task the robot could cooperate with other delivery robots for avoiding collisions or for giving up its turn in traversing dangerous work areas.

Comments and Symposium Question Answers

Coordination – How should the agent arbitrate/coordinate/cooperate its behaviors and actions? Is there a need for central behavior coordination?

In order to achieve more complex forms of interaction within a complex environment, an agent has to coordinate its actions. The necessity of coordination appears whenever conflicting situations appear. The case when situations and actions are not independent is actually the interesting case. The form of coordination depends on the task and on the interaction with the environment and does not necessarily have to be centralized.

This question implicitly assumes that the system has a behavior-based control. Behaviors represent responses to external stimuli from the environment. However, the way the agent behaves refers in general to the higher level appearance of the way the system responds over an observable period of time, and depends on the complexity of the environment. Behavior may be an emergent property of the system. In this case, the actions taken by the system may not have a clear correspondent in the overall behavior. The specific coordination or interdependence between actions is responsible for the overall behavior. Arbitration/coordination usually designates explicit control mechanisms for solving the interactions among actions.

The subsumption architecture has a primitive type of coordination. Coordination is ensured by the fixed priority ordering on the set of behaviors. The most important active behavior is the one whose actuator commands are executed.

A blackboard-based architecture enables process coordination based on the status of blackboard variables. In this way more complex forms of coordination are possible. Particular examples of coordination are sequentialization and parallelization of behavior activation.

Interfaces– How can human expertise be easily brought into an agent's decisions? Will the agent need to translate natural language internally before it can interact with the world? How should an agent capture mission intentions or integrate various levels of autonomy or shared control? Can restricted vocabularies be learned and shared by agents operating in the same environment?

Interaction with humans and other agents will play an increased role in the future development of autonomous agents. The language of any such interactions has to operate on entities common to the knowledge representation formalisms of the two communicating agents.

Natural language is neither necessary nor is it the most efficient means of communicating with a machine, although it is the most general one. Depending on the application task a restricted language could be not only valuable for run-time interaction between agents but essential for the completion of the task. The level of the communication language imposes lower bounds on the complexity level of the knowledge representation and reasoning capabilities of the agent. An important communication problem, inherent in natural language, is ambiguity. Humans could easily interpret requests from the agent but they should transmit to the agent only requests that can be represented unambiguously in the communication language.

Our application did not address any of these issues directly since human interaction was not required for completion of the robot's task. A restricted vocabulary however, would have been very useful in the human/robot interaction required during skill learning

and would have significantly accelerated this stage of program development.

Representation– How much internal representation of knowledge and skills is needed? How should the agent organize and represent its internal knowledge and skills? Is more than one representational formalism needed?

Pragmatics dictates that suitable knowledge representation (KR) paradigms depend on the task. However knowledge complexity and knowledge level enter the equation of the speed vs. memory tradeoff. This is particularly important for autonomous agents, characterized by exhaustible power supplies, other physical limitations and bounded resources. A more complex representation needs more memory and involves more complex decision making. The consequence is an increase in the response time, unless the "make the common case fast" heuristic is used in the design of the architecture. A simple example of the tradeoff between the generality of the representation and reasoning mechanisms and the overall agent efficiency is offered by our application. For example, map building is not feasible in a four minute task, but is the way to go if the robot is supposed to execute the same task many times, with a small turnaround time.

In moderately complex applications more than one representation formalism might be needed. Such an approach might improve system modularity and scalability. However one needs to convert knowledge, or partial solutions/results between representation paradigms which have different capabilities and computational characteristics. The form of the decomposition is an important question and one which is usually solved in a domain dependent way.

Structural– How should the computational capabilities of an agent be divided, structured, and interconnected? What is the best decomposition/granularity of architectural components? What is gained by using a monolithic architecture versus a multi-level, distributed, or massively parallel architecture? Are embodied semantics important and how should they be implemented? How much does each level/component of an agent architecture have to know about the other levels/components?

We have suggested that the architecture control kit (ACK) offers a flexible and efficient way of dividing up, structuring and interconnecting the computational capabilities of an agent. Such an approach is oriented towards the opportunistic activation of problem solving modules, defined to compete or cooperate. Problem solving power emerges based on the effects of processes corresponding to knowledge source activation records [4]. The granularity of decompositions is determined by the granularity of knowledge sources which is, in turn, under user control. Knowledge sources are defined as chunks of code written in a base language.

Performance– What types of performance goals and metrics can realistically be used for agents operating in dynamic, uncertain, and even actively hostile environments? How can an architecture make guarantees about its performance with respect to the time-critical aspect of the agent's physical environment? What are the performance criteria for deciding what activities take place in each level/component of the architecture?

It is almost impossible to come up with a general quantitative performance measure. Quantitative task dependent performance measures are important, but will always measure just certain solution aspects. Therefore, a good overall evaluation would have to take into account performance measures over several different evaluation directions on a multitude of environmental situations, ranging from easy to very difficult ones. Combining different performance measures is not an easy task either. Real world offers an environment with unexpected events or with unknown probability distribution of events, which determines a stochastic algorithm behavior. Good performance with high probability on any inputs is required. On the other side, the evaluation of failures may also be useful. Contrary to the common belief that we know what does not work, we miss rigorous analyses of experiments that do not work or in which solutions are very hard to come by.

Psychology– Why should we build agents that mimic anthropomorphic functionalities? How far can/should we draw metaphoric similarities to human/animal psychology? How much should memory organization depend on human/animal psychology?

Metaphoric analogies to human/animal psychology should offer alternative implementation means and ideas, and generate insights into the functioning of both artificial and natural designs.

Simulation– What, if any, role can advanced simulation technology play in developing and verifying modules and/or systems? Can we have standard virtual components/test environments that everybody trusts and can play a role in comparing systems to each other? How far can development of modules profitably proceed before they should be grounded in a working system? How is the architecture affected by its expected environment and its actual embodiment?

Simulation, when used, should offer a sufficiently complex environment. In this case, simulation is important from several points of view. First it offers the means to test the system on a huge diversity of inputs from a highly dimensional input space. This is recommended because of the stochastic nature of real environments. It affects software debugging, as experiments can be replicated, and system development. Second, simulation offers opportunities for parameter adaptation and even learning. However, simulation can never replace real world testing and evaluation. Different performance criteria may be generally needed for simulation and real world testing.

Learning– How can a given architecture support learning? How can knowledge and skills be moved be-

tween different layers of an agent architecture?

Without adaptation and learning mechanisms artificial intelligence systems are "inherently biased by the features and representation schemes defined by the system designer" [7].

I view adaptation as an automatic adjustment of control parameters to environmental conditions in order to make the agent more fit to the conditions in which it operates. Learning subsumes the gain or reorganization of knowledge, understanding and skills as a result of instruction or self-introspection and modification of behavior as a result of the experience in the interaction with the environment. Learning can itself be subject to evolution.

The learning problem is very hard in general. Computational learning approaches rely heavily on the representation formalism used. In simplified approaches learning takes the form of an optimization of parameters or generalization problem. We have attempted preliminary work on the influence of two types of learning mechanisms. The first is based on the idea of environment exploration. The agent takes advantage of its current path in the world in order to update a probabilistic grid-world model of the world. It has provisions for exploitative goals in case no other direct goals exist. A second type of learning, skill acquisition, is used mostly as a development aid to generating more complex behaviors.

Acknowledgments

The robot presented was built for a Mobile Robots seminar at the University of Rochester by the authors after an initial design by Randal Nelson and the authors. We want to thank Chris Brown, the mentor and organizer of the seminar, for the continuous encouragement he has given during this seminar and for the challenging competition organized. The competition rules have suggested to us their refinement into the current application.

References

[1] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1), March 1986.

[2] Rodney A. Brooks. Intelligence without reason. Technical Report A.I. Memo No. 1293, MIT, April 1991.

[3] Cristian Giumale. A rule-inference package for ai programming. In *Proceedings of the 7-th International Workshop on Expert Systems and their Applications*, Avignon, 1987.

[4] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3), 1985.

[5] John H. Holland. *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, second edition, 1992.

[6] Joseph L. Jones and Anita M. Flynn. *Mobile Robots, Inspiration to Implementation.* A K Peters, Wellesley, Massachusetts, 1993.

[7] Hiroaki Kitano. Challanges of massive parallelism. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 813–834. Morgan Kaufman, 1993.

[8] Henry Lieberman. A preview of act1. Technical Report AI Memo No. 625, MIT AI Laboratory, 1981.

[9] Henry Lieberman. Thinking about lots of things at once without getting confused: Parallelism in act1. Technical Report AI Memo No. 626, MIT AI Laboratory, 1981.

[10] Pattie Maes. A bottom-up mechanism for behavior selection in an artificial creature. pages 238–246. 1991.

[11] Pattie Maes. Behavior-based artificial intelligence. In *Proceedings of the Second Conference on Simulated and Adaptibe Behavior.* MIT Press, 1993.

[12] Tom M. Mitchell. Becoming increasingly reactive. In *Proceedings of the AAAI*, pages 1051–1058, 1990.

[13] Francesco Mondada, Edoardo Franzi, and Paolo Ienne. Mobile robot miniaturisation: a tool for investigation in control algorithms. *Third International Symposium on Experimental Robotics*, Kyoto, Japan, Oct 28-30, 1993.

[14] Justinian Rosca. Parallel knowledge processing - an sgi implementation. Technical Report unpublished report, University of Rochester, Computer Science Dept., 1993.

[15] Charles E. Thorpe. Mobile robots. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 2. John Wiley & Sons, Inc., second edition edition, 1992.

[16] Steve Whitehead and Dana H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, 1991.