A Modular System Architecture for Autonomous Robots Based on Blackboard and Publish-Subscribe Mechanisms

Somphop Limsoonthrakul*, Matthew N. Dailey*, Methee Srisupundit†,
Suwan Tongphu*, and Manukid Parnichkun†
*Computer Science and Information Management
†Mechatronics
Asian Institute of Technology
Klong Luang, Pathumthani 12120 THAILAND
Email: {somphop.limsoonthrakul, mdailey, mathee.srisupundit, suwan.tongphu, manukid}@ait.ac.th

***Abstract*—We present a system software architecture for mobile robots such as autonomous vehicles. The system achieves the goals of flexibility, maintainability, testability, and modifiability through a decoupled software architecture based on an asynchronous publish-subscribe mechanism and a blackboard object handling synchronized access to shared data. We report on two implementations using the proposed generic architecture and the POSIX real time API. The first implementation is for an autonomous vehicle using waypoint-based navigation, and the second implementation uses the same high-level modules but replaces the low-level hardware interfaces with a virtual reality simulation. Our experiments and an evaluation indicate that the architecture is suitable for a wide variety of control algorithms and supports the construction of testable, maintainable, and modifiable autonomous robot vehicles at low cost in terms of real-time performance.**

***Index Terms*—Software Architecture; Real Time Systems; Decoupling; Mobile Robot Control; Autonomous Vehicles.**

I. Introduction

Mobile robots such as intelligent vehicles need to transduce sensor measurements, infer the state of the world, plan future actions, and execute the plan, all in real time using limited compute resources. An appropriate software architecture is crucial for managing complexity, concurrency, and real time constraints. The last 20 years has seen a great deal of research in the AI robotics community on mobile robot control architectures emphasizing task flexibility and resolution of multiple conflicting goals. At the same time, the embedded systems engineering community has developed a variety of effective techniques for the implementation of robust systems composed of networks of control units executing tight control loops.

In this paper, we distinguish between the *robot control architecture* and the *system software architecture* and take a software engineering view of the design, implementation, and evaluation of a system software architecture that not only supports a variety of different robot control architectures, but also scores well in terms of software quality metrics for performance, testability, maintainability, and extensibility.

Deliberative control architectures range from simple control loops [1] to more complex layered architectures [2], [3] that decompose the robot's task into a hierarchy of subtasks organized into layers. Reactive methods [4], [5] combine many simple parallel short-circuit control loops that bypass higher-level planning entirely. Most modern control architectures, however, combine some aspects of both deliberative and reactive control [6]–[12]. Most of these systems use software architectures based on cooperating modules with messgage passing or procedue calls for communication and synchronization, but others use a shared database or *blackboard* for communication [13]–[15].

One potential problem with most of the existing system software architectures is that although modularity is a common design goal, they nevertheless introduce unnecessary dependencies between modules or objects, making it difficult to maintain the software and reuse modules across different projects.

In this paper, we propose, implement, and evaluate an architectural solution that attempts to reduce dependencies between modules to a bare minimum. Our approach combines an asynchronous messaging approach, in which sender and receiver are decoupled via a publish-subscribe paradigm, with a blackboard for updating and querying shared data. The implementation is based on the POSIX real-time API. We evaluate the architecture by first building a straightforward control system for an intelligent vehicle incorporating a variety of sensors, a deliberative waypoint-based planner, and an obstacle detection and avoidance module. We then extend the system by adding a virtual reality simulation for testing the perception, planning, and control software. We find that the approach produces efficient software with minimal inter-module dependencies. Although the control algorithm we chose for testing is fairly simple, it requires implementation of all of the important module communication patterns. The resulting software architecture would support the vast majority of control architectures that can be cast as loosely-coupled tasks cooperating through message passing and a shared blackboard.

The rest of the paper presents our architecture, implemen-

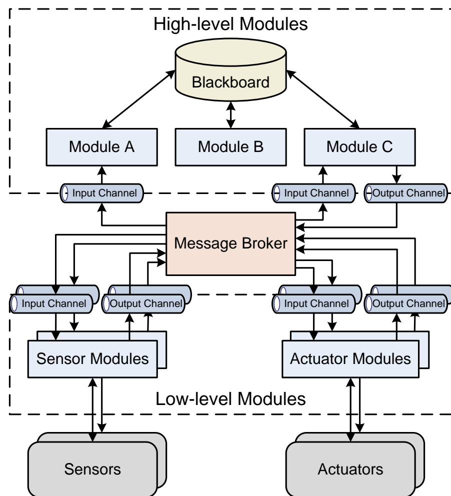

Fig. 1. Generic software architecture for an autonomous robot.

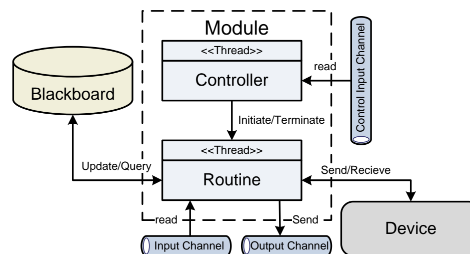

Fig. 2. Typical module structure.

tation experiments based on the architecture, and a critical evaluation of the architecture.

II. Software Architecture

Our proposed software architecture is shown in Fig. 1. The system is separated into two layers. The high-level layer contains task-specific perception, planning and control modules, and the low-level layer serves as an interface between the hardware (sensors and actuators) and the high-level layer. The low level modules simply execute the commands published by high-level modules, receive and process sensor data, and publish processed sensor data for consumption by high-level modules.

Within the layers, the software is structured into independent modules, each running one or more local threads. The modular structure allows for flexible construction of systems for specific vehicles by including and configuring an appropriate set of modules. The low-level layer modules depend on the hardware platform, but can be easily replaced with a new set of modules for a new hardware platform without affecting high-level modules. Tasks in the high-level layer are also separated into modules so that algorithmic changes in one module have little or no effect on other modules.

To allow the modules to share data and synchronize with each other, we use two main mechanisms: *publish-subscribe* messaging and a shared *blackboard*. These mechanisms are described in detail later in this section.

A. Module Structure

For flexibility, ease of implementation, and testability, we decompose perception, planning, and control tasks into many simple tasks. For example, localization with a GPS sensor could be decomposed into two modules: 1) *GPSReader* module responsible for receiving and processing messages from the GPS sensor and 2) a *Localization* module responsible for filtering raw sensor readings and updating the vehicle's state. With this decoupled structure, the Localization module does not need to know how to connect and get data from the GPS sensor, so changing the sensor or communication protocol has no effect on Localization.

The typical module structure is shown in Fig. 2. Most modules need to export commands instructing them to start, stop, or suspend processing. To provide this functionality, we split the module into two main threads: a *Controller* thread and a *Routine* thread. The module's task or function, possibly involving communicating with hardware devices, producing or consuming messages, and interacting with the blackboard, is executed in the Routine thread. The Controller thread initiates and terminates the Routine thread as necessary. Control signals are passed to a module by inserting messages into a control queue, either directly (when point-to-point communication is most appropriate) or via publish-subscribe (described in the next section).

In most cases, modules should not communicate with each other directly; they should communicate by publishing and consuming messages or by interacting with the blackboard. These approaches eliminate coupling between modules, since neither the producer nor consumer of information needs to know about each other.

B. Publish-Subscribe Mechanism

One of our main goals is to enable the flexible construction of vehicle control systems by composing existing sets of modules. Since dependencies between modules mean that one module cannot be compiled, linked, or executed without the other, we must minimize dependencies to the extent possible. Dependencies are minimized when one module is able to notify other modules about important events without knowing the actual recipient list. This requires an event-driven system in which event consumers register for (subscribe to) asynchronous notification of specific event types and event producers simply generate (publish) events as necessary and continue processing. The publish-subscribe messaging pattern is perhaps the most straightforward way to implement this behavior.

As shown in Fig. 3, the publish-subscribe pattern consists of 3 main entities: the publisher, the broker, and the subscriber. Events (messages) are categorized into types with corresponding message formats. The publisher produces and publishes messages that are consumed by the subscriber. The broker is a message router which, among other duties, forwards each

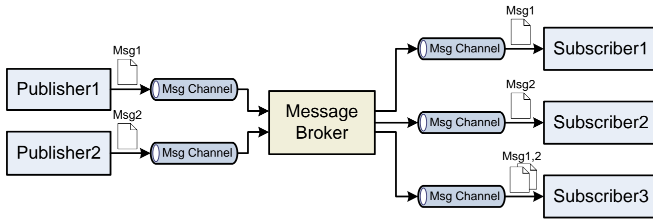

Fig. 3. Publish-subscribe messaging pattern.

message from a producer to zero or more subscribers, each of which must register for the particular message types it would like to receive. Publishers and subscribers are decoupled through the use of *message channels* that are set up as part of the configuration and initialization of the system. Since the publisher and subscriber only need to know the addresses of their respective output and input message channels, they are only coupled to each other by message format.[1] The broker acts as an intermediary, listening to every module's outgoing message channel and, according to the message type, forwarding messages to each subscriber's incoming message channel.

Continuing with the vehicle localization example from section II-A, in which we have GPSReader and Localization modules, we can apply the publish-subscribe pattern to establish communication between them. At initialization time, we first define a message type for GPS data and create an outgoing message channel for GPSReader and an incoming message channel for Localization. Next, the outgoing message channel for Localization needs to be registered with the message broker. We recommend setting up all message channels and subscriptions in a main initialization routine, to prevent the modules from being coupled to the message broker. At run time, when GPSReader receives data from the GPS sensor, it creates a message and inserts it into its outgoing channel, without knowing what modules will end up retrieving the message. The broker, which listens the the GPSReader's outgoing message channel, receives the message and forwards it to the to Localization's incoming message channel. Thus, Localization receives asynchronous GPS updates whenever they become available, without knowing anything about the sender. Likewise, other modules requiring GPS updates can receive the same data by subscribing with the broker.

C. Blackboard Mechanism

Not all inter-module communication patterns can be handled by an asynchronous publish-subscribe mechanism. For example, a planning module might need to immediately know the current best estimate of the vehicle's position and speed at a particular point in the near future. It could, say, make a synchronous request-response call to the localization module, but handling such synchronous requests would complicate the design of the localization module, which otherwise would only have to repeatedly listen on its input message channel for incoming sensor data, update its estimate of the state, and publish the resulting estimate.

To solve this problem and to provide a general mechanism for sharing global data, we use a blackboard. Our blackboard is a coarse-grained singleton object that encapsulates and synchronizes access to all of the shared world state information needed by the high-level modules to perform their duties. The synchronization method is a standard shared read lock/exclusive write lock solution to the readers-writers problem. Our current implementation is non-preemptive and allows priority inversion, so designers of the write routines need to ensure that all updates are non-blocking and complete in constant time.

The main advantage of the blackboard is that it provides a simple mechanism for sharing data between modules without coupling those modules. For example, the localization module can update the vehicle's position at any time, and the planning module can get the current vehicle position at any time, but the two modules need not know about each other.

This leads to the main disadvantage of the blackboard, namely that many modules end up coupled to the blackboard itself. This coupling is further complicated by the fact that in general, the blackboard API will be task specific. Possible solutions to extreme cases of these application-specific dependencies could include 1) refactoring a large incoherent blackboard into multiple blackboards for different functional areas of the control system or 2) using standardized database technology. Unfortunately, if two modules in a design need to share data, there are few practical choices other than a shared database to decouple the modules. Fortunately, however, we find that in practice, the API for a functional area tends to stabilize quickly after the first version.

III. Implementation

In this section we describe the implementation of the system architecture's infrastructure, application of the approach to the construction of a basic control system for a real automobile, and then, to measure some of the architecture's quality attributes, especially extensibility, we integrate the control system with a virtual reality simulation.

A. Control Design

The vehicle's task is to follow a predetermined path defined by a sequence of waypoints. Along the way, it is required to avoid obstacles on the road and obey traffic regulations as directed by a fixed set of traffic signs. Our design uses five main high-level modules: localization, obstacle detection, traffic sign recognition, planning, and control.

1) Localization: The current simple design uses a GPS sensor and an electronic compass for localization. The two sensor modalities are integrated with control actions to recursively estimate a three degree of freedom state using an extended Kalman filter. State estimates are published to the blackboard.

[1] If complete decoupling of publisher and subscriber is desired, it is possible to add transparent message format translation capabilities to the message broker.

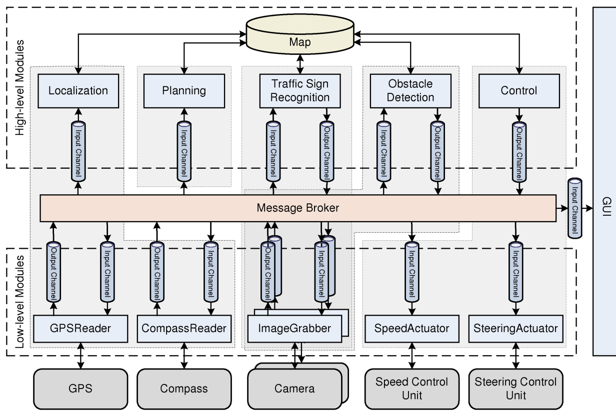

Fig. 4. Specific system software architecture for Experiment I (Little-MEC II).

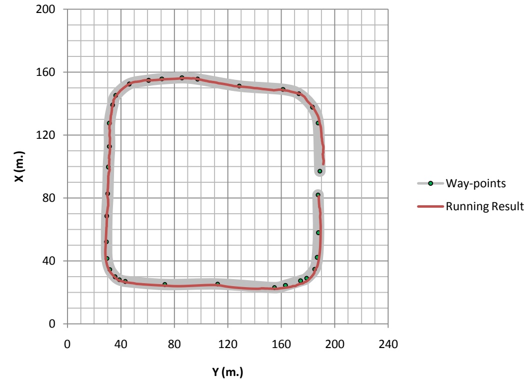

Fig. 5. Tracking results for Experiment 1 (Little-MEC II).

2) Obstacle Detection: We use a front-facing monocular camera to detect obstacles. When possible obstacles (deviations from the background road color) are detected, their positions in the vehicle coordinate frame are posted to the blackboard.

3) Traffic Sign Recognition: We use a second monocular camera to detect traffic signs using color and shape information, then perform template matching to categorize signs. Based on the position and size of a sign in the image, we estimate sign positions in the vehicle coordinate frame and post the resulting signs to the blackboard.

4) Planning: The system repeatedly updates the status of way-points that have been reached and decides which to pursue next. Additional waypoints are added and deleted as necessary to avoid obstacles and follow the instructions of traffic signs.

5) Control: Speed setpoints are specified for each waypoint as part of the predetermined map. We determine the steering setpoint using fuzzy logic based on the orientation error, differential orientation error, and distance to the next waypoint.

B. Experiment I: Little-MEC II

We implemented the architecture and control algorithm just described on Little-MEC II, a Mitsubishi Galant with a 1.8 L engine and a four-speed automatic transmission. We equipped the system with a GPS sensor and electronic compass communicating via RS-232, two cameras communicating via IEEE-1394, and a 3.0 GHz Pentium Core 2 Duo with 1 GB of RAM running GNU/Linux (Ubuntu 7.10).

We developed the architectural infrastructure and control software in C. Each module is implemented by one or more POSIX threads (POSIX threads are kernel threads on GNU/Linux). We implemented the blackboard's synchronization mechanisms using POSIX mutexes and condition variables. We implemented the publish-subscribe mechanism using POSIX message queues. Since POSIX queues are point-to-point, the message broker intermediary is necessary to provide registration and multiple-destination routing facilities.

As shown in Fig. 4, the Little-MEC II system consists of 11 modules represented by blue boxes. All of the communication between layers is via the message broker using the publish subscribe mechanism. The shaded boxes in the system architecture represent groups of modules that work together on related tasks.

The Map object shown on the top of the diagram is a blackboard used by the cooperating high level tasks. It consists of many data structures, such as sequences of way-points, the car state history (position, orientation, and speed), and the positions of obstacles and traffic signs. We allow multiple concurrent readers and give writers higher priority than readers to prevent writer starvation. In our initial experiments, we have only needed immediate writes and polling reads, but asynchronous blackboard event handling would be straightforward to implement if necessary.

The GUI module is responsible for initialization and allows users to control (start/stop/suspend) threads running in each module. The GUI module uses point-to-point message passing to implement control actions. Since the GUI is necessarily coupled to all modules via the initialization and control procedures, its implementation is simplified without any adverse architectural effect.

As a first system-level test, we set a path for Little-MEC II to navigate along a road around a football field at a speed of 5 to 10 km/hr. The result is shown in Fig. 5.

C. Experimemt II: Simulation

As a test of the flexibility and extensibility of the proposed architecture, we modified the system described in section III-B by replacing the real hardware with a virtual reality simulation.

The modified system architecture is shown in Fig. 6. The simulator consists of two main components, the vehicle model and the camera simulator:

1) CarModel: The vehicle model is an active blackboard encapsulating the automobile's geometry and state. We use a non-slippery bicycle model for the dynamics assuming that the mass of the vehicle is a point at the center of the rear axle. The module maintains the current state of the car (position, orientation, and speed) while the simulation is running, performing updates at 1000 Hz.

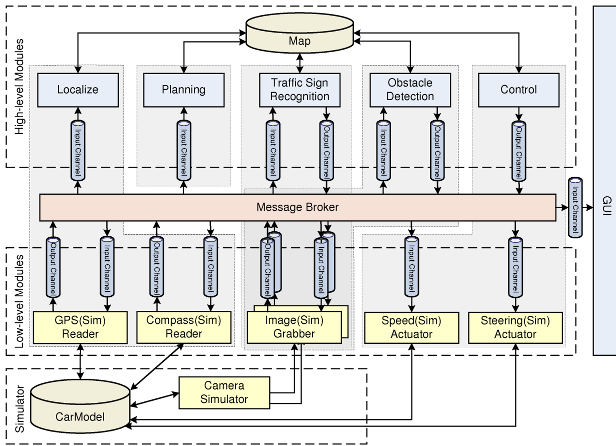

Fig. 6. Specific system software architecture for Experiment II (virtual reality simulation). Switching between the real world and the simulation is a simple matter of configuring a different set of low-level modules at initialization time.

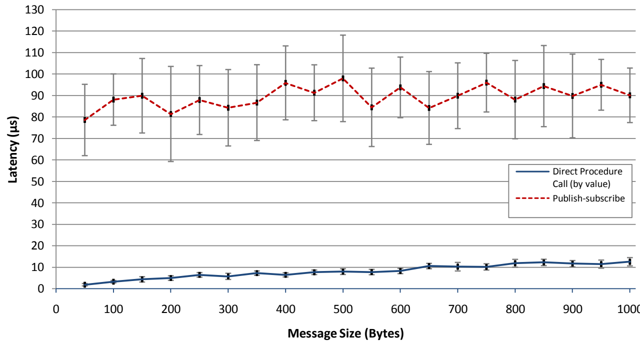

Fig. 7. Latency of message passing using publish-subscribe compared to direct procedure calls.

2) CameraSimulator: We use the OpenGL library to render 3D scenes based on a background environment model and the car state produced by the CarModel, using the same camera positions as the real car. When called, the module creates a 3D model of the vehicle and the surrounding environment, and according to the car state (position and orientation), it renders the images and passes them to the Image(Sim)Grabber.

Integrating the simulator required changing a few modules. In particular, the modules in the low-level layer that interface directly with sensors and actuators needed to be replaced with new modules. The yellow boxes in Fig. 6 are the modules we had to replace or modify; they are restricted to the low-level layer. The simulated sensor reader modules run their own threads and issue queries to the CarModel blackboard, and the actuator modules also update the CarModel module.

IV. Architecture Evaluation

In this section, we critically evaluate the proposed system software architecture. We focus on performance, testability, maintainability, and modifiability.

A. Performance

To evaluate the performance of our system, we compare the latency of our inter-module communication mechanisms with that of direct procedure calls.

In the case of the blackboard, modules cooperate with each other via shared memory. Theoretically, the communication latency for the blackboard is just the amount of time used to update memory plus the amount of time the recipient thread waits to access the data. These are exactly the same steps that are required for direct module-to-module communication, so as long as starvation is prevented, the blackboard mechanism does not increase the latency of inter-module communication.

Our publish-subscribe mechanism, however, uses intermediaries (POSIX message queues and the message broker) to route messages between modules. This surely increases latency of inter-module communication compared to direct procedure calls. To determine this cost, we performed a simple experiment to empirically compare the communication latency of our publish-subscribe mechanism compared to direct procedure calls. We built a testing system consisting of 3 modules: a publisher, a subscriber, and the message broker. We constructed test messages consisting of a message header, a fixed-size character string representing the message payload, and a timestamp. To determine the latency, we set up the publisher to repeatedly set the timestamp then send the message, and we set up the subscriber to retrieve the sent message then compare the message timestamp to the current timestamp. We varied the message payload from 50 to 1000 bytes.

The results of the comparison are shown in Fig. 7. The overhead of the publish-subscribe mechanism, while significant, is on the order of 100 microseconds on our test system. Since this is much faster than the sensor, actuator, and control loops (our sensor loops run at 100 Hz), it is manageable, so long as no more than a few messages are passed per control loop iteration. This will always be the case as long as the modules are coarse grained. The architecture thus satisfies the real-time demands of the typical autonomous robot application.

B. Testability

Our architecture is explicitly designed for testability. The restriction of dependencies between modules means we can very easily test at the unit, integration, and system levels. Unit testing the low-level modules only requires the module itself, the hardware device the module is communicating with, some means to generate test input messages for the module, and some means to examine and determine the correctness of the resulting output messages. Unit testing the high-level modules only requires the module itself, a means to generate test input messages for the module, a means to examine the module's output messages, and a test fixture for the blackboard's initial state.

We find that the modular and decoupled nature of the system makes it easy to construct integration tests that combine subsets of the modules to test the end-to-end data flow.

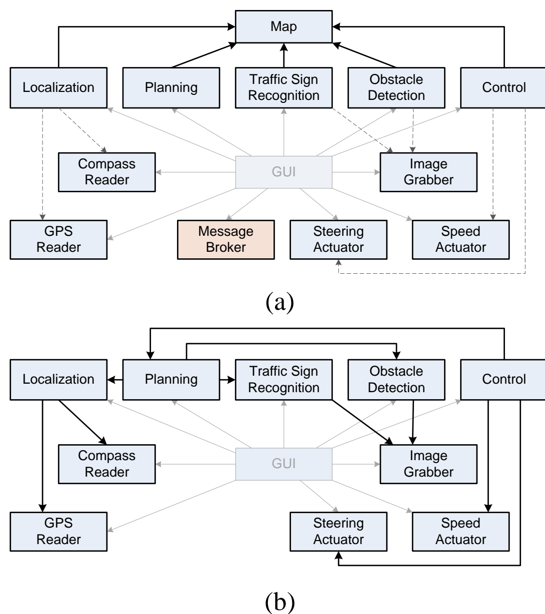

Fig. 8. Dependency diagrams with and without publish-subscribe and blackboard mechanisms. (a) Dependency diagram with publish-subscribe and blackboard mechanisms. (b) Dependency diagram that would be obtained if modules communicated directly. Solid lines represent dependencies between modules, and dashed lines represent message format dependencies.

Finally, since the system is flexible and straightforward to extend with the virtual reality simulation described in section III-C, it is easy to construct specific full-system tests without the safety issues of testing experimental software on real vehicles.

C. Maintainability

The goal of maintainability is achieved by systems in which changes to the software remain local without affecting other parts of the system. Modularity is one way to ensure that changes are local. Another indication of maintainability is the number of dependencies between modules. To assess the level of dependency in our system, we compare, in Fig. 8, the dependency diagram for the system using publish-subscribe and the blackboard, to a system using direct module-to-module communication.

From the figure, we see that the number of dependencies between modules, including dependencies on the blackboard, falls from 10 to 5. As previously explained, the modules are still slightly coupled by message formats, but this coupling can also be eliminated through message translation in the message broker. The decreased number of dependencies will very likely lead to more maintainable systems in the long run.

D. Modifiability

We believe that following the principles of modularity and dependency minimization in architecture design leads to systems that are easy to modify and extend. This has already been proven in section III-C, in which we show that it is straightforward to modify an existing system based on our architecture by replacing a subset of the modules without affecting the rest of the system at all.

V. Conclusion

We have presented a system software architecture for mobile robots that supports the flexible construction of efficient, testable, and modifiable autonomous robot vehicles. A series of two implementation experiments and an evaluation indicate that the approach is quite promising. In future work, we plan to build a wider variety of more complex mobile robots on top of the basic platform and refactor as necessary to achieve flexibility and reusability.

Acknowledgments

This research was supported by Seagate, the Thailand National Electronics and Computer Technology Center (NECTEC), the Royal Thai Government, and the Thai Robotics Society. We are grateful for the hardware support provided by the AIT Intelligent Vehicle team. Chaianun Damrongrat provided valuable comments on this work.

References

[1] T. Lozano-Pérez, “Foreward,” in *Autonomous Robot Vehicles*, I. Cox and G. Wilfong, Eds. New York: Springer, 1990.
[2] A. Elfes, “Sonar-based real-world mapping and navigation,” *IEEE Journal of Robotics and Automation*, vol. 3, no. 3, pp. 249–265, 1987.
[3] C. Laugier, T. Fraichard, P. Garnier, I. Paromtchik, and A. Scheuer, “Sensor-based control architecture for a car-like vehicle,” INRIA, Tech. Rep. AI Memo No. 1461, 1998.
[4] R. Brooks, “A robust layered control system for a mobile robot,” *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.
[5] G. Wasson, D. Kortenkamp, and E. Huber, “Integrating active perception with an autonomous robot architecture,” *Robotics and Autonomous Systems*, vol. 26, pp. 325–331, 1999.
[6] P. Althaus and H. Christensen, “Behavior coordination in structured environments,” *Advanced Robotics*, vol. 17, pp. 657–674, 2003.
[7] R. Simmons, “An architecture for coordinating planning, sensing, and action,” in *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*. U.S. Defense Advanced Research Projects Agency, 1990, pp. 292–300.
[8] ——, “Structured control for autonomous robots,” *IEEE Transactions on Robotics and Automation*, vol. 10, no. 1, pp. 34–43, 1994.
[9] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, “An architecture for autonomy,” *The International Journal of Robotics Research*, vol. 17, no. 4, pp. 315–337, 1998.
[10] M. Lindstrom, A. Oreback, and H. Christensen, “Berra: A research architecture for service robots,” in *Proceedings of the 2000 IEEE International Conference on Robotics & Automation*, 2000, pp. 3278–3283.
[11] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, “The CLARAty architecture for robotic autonomy,” in *Proceedings of the 2001 IEEE Aerospace Conference*, 2001, pp. 121–132.
[12] T. Fraichard and P. Garnier, “Fuzzy control to drive car-like vehicles,” *Robotics and Autonomous Systems*, vol. 34, pp. 1–22, 2001.
[13] S. Schneider, M. Ullman, and V. Chen, “Controlshell: a real-time software framework,” in *Proceedings of the 1991 IEEE International Conference on Systems Engineering*, 1991, pp. 129–134.
[14] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti, “The saphira architecture: a design for autonomy,” *Experimental & Theoretical Artificial Intelligence*, vol. 9, pp. 215–235, 1997.
[15] S. Shafer, A. Stentz, and C. Thorpe, “An architecture for sensor fusion in a mobile robot,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, 1986, pp. 2002–2011.