INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Emergent Architectures:
A Case Study for Outdoor Mobile Robots

Jay Gowdy

CMU-RI-TR-00-27

Submitted in partial fulfillment of the
requirements for the degree of Doctor of
Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

November 1, 2000

Keywords: Software architectures, outdoor mobile robots, system integration, reuse.

Carnegie Mellon

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Thesis

Emergent Architectures: A Case Study for Outdoor Mobile Robots

Jay Willard Gowdy

Submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the field of Robotics

ACCEPTED:

Charles E. Thorpe — Thesis Committee Chair — 11/13/00 — Date

Matthew T. Mason — Program Chair — Dec 12, 2000 — Date

James H. Morris — Dean — 1/5/01 — Date

APPROVED:

Mark S. Kamlet — Provost — 1/5/01 — Date

Abstract

Software reuse is a key issue in any long term software engineering endeavor, such as the ongoing development of robotics systems. Existing approaches to software reuse involve fixing part of the software landscape as a constant foundation to build upon. Some approaches fix the software architecture, i.e., how data flows through the system, and then allow the user to swap components in and out. Other approaches define a suite of stable components and interfaces which can be mixed, matched, and extended. All of these approaches assume that the key to software reuse is pervasive and long lasting standards. Unfortunately, in a young domain such as mobile robotics both software components and software architectures are items of research, and thus are in constant flux.

This dissertation proposes the replacement of global, system-wide, permanent standards with local, transient standards in the form of reconfigurable interfaces. These interfaces are not simply libraries of system calls, but contain mediators and adapters which stand between a software module and the system architecture, providing a module with a locally stable view of the system in which it resides, translating the module's requests and notifications into the protocols and data flow of that particular architecture. This system of mediation facilitates the emergence of a system architecture from the current needs and requirements of the tasks, rather than forcing the a priori adoption of any single architecture.

To demonstrate the feasibility of this approach, two very different robotic tasks were examined: first the implementation of a dedicated, high-speed road following system under Carnegie Mellon's NavLab project, and second the implementation of a multi-modal military reconnaissance vehicle for the Unmanned Ground Vehicle program. Radically different architectures and components emerge from the requirements of these two tasks and the current capabilities of the component modules, but it is demonstrated that reconfigurable interfaces allow the moving of critical code between the two without even recompilation. The mediating abilities of reconfigurable interfaces are provided without requiring any pervasive, long-lasting communications standards while only requiring the execution overhead of a few pointer dereferences.

The emergent architecture approach acknowledges the inevitable fluidity of both robotic components and system architectures. The reconfigurable interfaces act as a buffer for change flowing from the bottom up or change flowing from the top down, thus allowing the concurrent development of modules and architectures. Reconfigurable interfaces enable a natural, incremental, and empirical approach to building robotic systems in which the ultimate system architecture is free to emerge from the current task requirements and component capabilities, and the component developers are free to make improvements in their modules with out an overriding regard to the systems in which they reside.

Acknowledgments

The work of a systems integrator can never really stand alone: there is always a large cast of people without whom a system will never come together. I would like to thank the people in the Robotics Institute who provided me with amazing software to integrate such as Martial Hebert, Omead Amidi, Tony Stentz, Dean Pomerleau, and Todd Jochem. A special thanks to Jim Frazier for keeping the Navlabs going, on the road and in the hibay. I would also like to thank the people at Lockeed Martin for being willing to be inspirations and guinea pigs for my ideas of remote system integration, especially Matt Morgenthaler, Mark Rosenblum, Alan Dickerson, and Betty Glass.

Most of all, I would like to thank my advisor, Chuck Thorpe, first for giving me my first job, and second, for turning my first job into a career. Without his wise leadership, gentle guidance, and amazing patience, little of my work would have been possible.

This work was funded in part by DARPA grants DACA76-89-C-0014 ("Perception for Outdoor Navigation") and DAAE07-90-C-R059 ("Unmanned Ground Vehicle Systems").

Table of Contents

List of Figures

Chapter 1 Introduction

1.1 Robots and architectures

Most large, long term mobile robot systems start with the software architecture. All of the important experts sit in a room and define the architecture, and then the implementation details are worked out over the course of the project. The resulting architecture is intended to be **the** guideline for how the components of the system work together for the remainder of the project.

The robot software architecture that the group of designers seeks is the framework that merges individual components into a coherent system. Usually "architecture" refers to the flow of information and control, and does not directly address the details of hardware design, although the hardware and architecture will certainly influence each other. Architectural concerns include which sensors are connected to which modules; which actuators are controlled by which modules; how information and control flow, including low-level mechanisms and high-level protocols; what centralized resources are made available to all modules; procedures for incremental building and testing; conflict resolution; human interfaces; resource management; real-time guarantees; control stability; etc. The architecture is not concerned with the internals of modules, such as the details of how a particular image interpretation system works or the gains of control loops internal to a single module.

An underlying motive of our roomful of mobile robot system designers is often to produce **the** mobile robot system architecture, not just a mobile robot system architecture. The drive for a standard architecture is understandable, since widespread standards would ease the long term development of projects and sharing of components between projects.

Unfortunately, the domain of "mobile robots" is broad enough that no one generic mobile robot architecture has yet addressed every possible mobile robot task. For example, a fully autonomous legged planetary robot must make statically stable, slow, "optimal" moves combining and deliberating on information from many sources while a semi-autonomous road traversing system drives down a road at 25 to 60 miles per hour reacting to sensed road information, obstacle information, and user intentions and guidance. It is highly unlikely that the same system architecture will even suffice for the two tasks, let alone that the same system architecture will provide the best satisfaction of the wildly different requirements of the two tasks.

In fact, the goal of **the** software architecture for a particular mobile robot application may also be misguided. First, it is almost impossible to guess all of the requirements of an application before that application has been investigated. More importantly, the requirements are constantly changing. For example, since the Navlab, a Chevrolet van modified for computer control [Thorpe, 1990], was first introduced in 1986 there have been huge advances in the capabilities of our computing and algorithms. This advance in capability has been more than matched by an advance in ambition. The demands of the Navlab's tasks quickly outstripped the capabilities of the initial 1986 architecture.

1.2 Robot trucks and cars at the Robotics Institute

My approach to architectures and system integration for mobile robots flows from the Carnegie Mellon Robotics Institute's thirteen year experience in the Navlab program and my six years of experience as the chief systems integrator.

Figure 1 shows a portrait of the Robotics Institute test bed vehicles that are part of the Navlab family. The original Navlab, funded by the Autonomous Land Vehicle program, was fairly limited. It had a specialized hydraulic transmission which gave the vehicle a top speed of less than 20 miles an hour, but which allowed extremely precise speed control at very low speeds on smooth

terrain such as roads and paths [Thorpe, 1990]. This precision was necessary because the algorithms running on the original constellation of Sun 3's gave a maximum safe speed of roughly 10 cm/s. Eventually, algorithmic and computing improvements made the physical capabilities of the Navlab, i.e., its limited speed and low ground clearance, the limiting factor in advancing the research.

The Navlab 2 is a converted military HMMWV introduced in 1991 as part of the Unmanned Ground Vehicles (UGV) program. The focus of the UGV program is to demonstrate that autonomous vehicles can be useful for military tasks such as reconnaissance and surveillance. Whereas the Navlab 1 had a ground clearance of about 4 inches, the Navlab 2 has a ground clearance of about 16 inches, which means that we could do much more realistic research in cross country navigation. The Navlab 2 also has a maximum speed of 60mph, which allowed us to continue the road following research at higher speeds. The Navlab 2 has driven autonomously over kilometers of unstructured terrain and on highways at highway speeds. The Navlab 4, a HMMWV converted for autonomous operation in 1994, is even more specialized for cross country experiments and allows us to investigate multiple cooperating vehicles working together on cross country navigation tasks.

Navlabs 3 and 5 are specialized for on-road navigation [Jochem et al., 1995]. We have built these vehicles under Department of Transportation programs which envision using automated vehicles to make driving safer, less tedious, and more efficient. Instead of being special research or military vehicles, these test beds are normal passenger vehicles modified for computer control. The Navlab 3, a Honda Accord, served as a prototype for the Navlab 5, a modified Pontiac Transport minivan built in 1994. The Navlab 5 has achieved speeds of 90 m.p.h. under computer control on a closed track and has driven across the country, 2850 miles, with the computers steering 98.2% of the time.

The impact of the Navlab project extends beyond Carnegie Mellon. Technology developed at Carnegie Mellon played a large part in the working of the integrated demonstrations of UGV technology done by Lockheed Martin in Denver [Munkeby and Spofford, 1996]. These integrated demos are done on multiple vehicles (see Figure 2) using many of the algorithms and architec-

Figure 1 The Navlab family portrait: From 1 on the far left to 5 on the far right

Figure 2 Lockheed Martin UGV's

tural tools developed at the Robotics Institute, as well as algorithms developed at 52 other locations.

Throughout the Navlab experience at the Robotics Institute there has been one constant: change. There have been incremental changes, such as upgrading of computing or the switching of one road following algorithm to another, and there have been large qualitative changes, such as the switching between the cross country tasks required by the UGV program to the highway navi-

gation tasks required by the Automated Highway System (AHS) program or the delivery of code from Carnegie Mellon to a site thousands of miles away to developers with different goals and requirements. Many of these changes have required changes in the system architectures, i.e., in how modules interconnect and communicate.

1.3 Architectures and development

In our mobile robot efforts there have been two distinct groups of workers: component developers and system developers. Component developers focus primarily on components such as road followers or strategic planners. They are focusing on specific algorithms and often have their own agendas involving research and publication of results specifically having to do with their own components. System developers focus primarily on systems, which are the result of organizing components and the flow of data between components to solve some bigger problem. System developers in our mobile robot efforts have focused on concrete, periodic demonstration of capabilities of integrated systems that would be impossible for any one component.

The success of both system development and component development hinges on the architecture. The result of system development, i.e., the component organization and data flow, is my very definition of an architecture. Component developers must build their components within some architecture in order to show the applicability of their component. Unfortunately, system developers and component developers have fundamentally conflicting ideals for how architectures and components should interact.

In a component developer's ideal world, the architecture is fixed and revolves around the component. In this ideal architecture the topology of the system is centralized, with data flowing to and from the component in a fixed manner. The only time this architecture should change is when in the course of developing the component a new requirement or capability is discovered. In addition, the implementation of the architecture should be tuned to best showcase the working of the component.

In a system developer's ideal world, all of the necessary components are well known, catalogued, and standardized. Building a system should be like reaching into a toy box and picking the right blocks and assembling them in the right ways. Then the architecture, i.e., which compo-

nents need to be selected, how they are connected, and how data flows between them, can be driven solely by the current task demands.

Unfortunately, in the domain of mobile robotics, neither group lives in its ideal world. The components of a mobile robot system are research projects unto themselves. It is impossible to completely predict what kind of changes will take place in a component. As the component developer works on the component and tries different approaches, the demands the component places on the architecture it resides in will change, i.e., over time a component may require different levels of system performance to work optimally, and may even require qualitatively different kinds of information. Thus, changes to an architecture, i.e., the manner in which components are organized and communicate, can bubble up from fundamental changes in the components themselves. In addition, the optimal organization of components into a coherent whole is as much a matter of research as the components themselves, and thus as prone to unpredictable change as the components themselves. System developers must be free to experiment and evaluate different configurations of components and different methods of combining information from components. This freedom means changes in the architecture will filter down from the externally imposed (and unfortunately changeable) task requirements and empirically discovered implementation constraints.

1.4 Thesis

- *A fixed, a priori reference architecture should not be the basis of a system.*

  Every element of a mobile robot system is undergoing constant research and development, from the control algorithms used to move the vehicle to the high level planners used to decide the system goals and priorities. The system software architecture is no exception: it needs to respond to the changes in the components to best fit the needs of what is currently in the system, as well as being a subject of research and development in and of itself. If the architecture is not free to adapt and change, the system will suffer.

- *When the architecture changes, it must be easy to move components to the new architecture.*

  The components that make up a system represent hundreds of thousands of lines of code

and years of research. If they must be substantially changed or rewritten when the architecture changes, then the architecture will most likely remain fixed and unresponsive to changes in requirements and abilities until the architectural shortcomings become overwhelming.

- *The basis of an evolvable system is a set of toolkits which isolate the individual components from the current instance of the architecture.*

  If a component is isolated from the architecture in which it resides, and that architecture changes, that means that reintegrating the component into the new architecture is not a large task. Thus, the capital investment required to change the architecture has been drastically reduced, and the architecture is free to adapt to the current requirements of the domain and task and the current capabilities of the hardware and software.

I present two benchmark applications which I have built and tested on real vehicles, the primary one being a multi-modal cross country navigation system and the secondary one being an urban road traversal system. I show how two fairly different architectures emerge from the requirements of the two different tasks given the current capabilities of the software and hardware. I then show how the architectural toolkits make it possible to move components from one architecture to another with no changes in the actual code.

My approach to architectures allows for a ruthless pragmatism with the individual components as well as the architecture as a whole: Components and functionalities that are not needed for the testing of the task as it currently stands should be dropped. For example, if there is no current pressing need for an extensive, general reasoning system to successfully perform a task then don't include one. If in later development and experimentation a reasoning system proves useful and necessary, then the architecture should adapt to include it, rather than including a possibly unnecessary component from the start. By building an expectation of change into my systems, I know that I can afford to leave things out that may or may not be needed later, rather than including everything that might possibly be needed from the beginning.

1.5 Emergent analogy

Artificial intelligence was once envisioned as the center of an intelligent, autonomous mobile robot. In the traditional AI approach, intelligence is defined as a centralized structure manipulating symbols in a world model. The system perceives the world through a set of "world to symbol" converters, which observe the world and convert their findings into symbols. The central intelligence can then incorporate the new symbolic knowledge of the world into a world model which can be manipulated and analyzed to determine the next action of the system. The system becomes more capable by adding more "world to symbol" converters for interpreting the world, more "symbol to world" converters for effecting the world, and by adding more sophistication to the symbolic manipulation and information processing [Simon and Siklossy, 1972].

Unfortunately, systems built using this approach did not live up to their promise. They did not work well in the dynamic, messy real world, especially once the robots left the laboratory for the outdoors. In an extreme reaction to initial failures of the centralized intelligence approach, the claim was made that there is no need for an a priori definition of where the intelligence of a system abides, in fact, there should be no a priori definition of what intelligence is [Brooks, 1991]. This reactive approach proposes that intelligence should "emerge" from the rigidly formalized interaction of a set of mindless behaviors in continuous contact with the real world.

The analogy that I propose and use is a similarly "emergent" architecture. In emergent intelligence, intelligent action emerges from the interaction of behaviors working in the real world without an explicit center of intelligence. An emergent architecture results from the interaction between the task requirements, hardware capabilities, and a set of software tools without an explicit inflexible a priori plan. A different task or even a different set of hardware can mean a different overall software architecture. The architecture must fit the task and the capabilities of the system. As the task and capabilities evolve, the architecture must be able to evolve as well.

1.6 Toolkits for emergent architectures

The basis of an emergent architecture is a toolkit that isolates the modules, which are the real meat of the system, from the architectures in which they reside. This isolation means that a module in which years of effort are invested can move from task to task, possibly even from domain to

domain, without having to be reengineered for each different architecture. Underlying the toolkit are reconfigurable interface objects that modules actually interact with. These objects may or may not correspond to physical objects in the system. For example, we know our road following module must interact with a "controller" object by sending it actuator commands, i.e., the steering direction. The implementation of this object varies from architecture to architecture. In one architecture the object simply prints these commands to the screen, in another the steering command actually goes to a real vehicle controller, and in another the single commanded arc gets converted into a set of arc "votes" to be fused with other sensorimotor modules (Figure 3). The road following module just "sees" the reconfigurable controller object, and doesn't have to deal with the issues of being in one architecture or another.

Chapter 2 goes into the details of how to develop interface objects for individual modules. I present a set of tools and templates which help in building interface objects that can be reconfigured at run time, allowing a module to move between certain architectures without even recompiling.

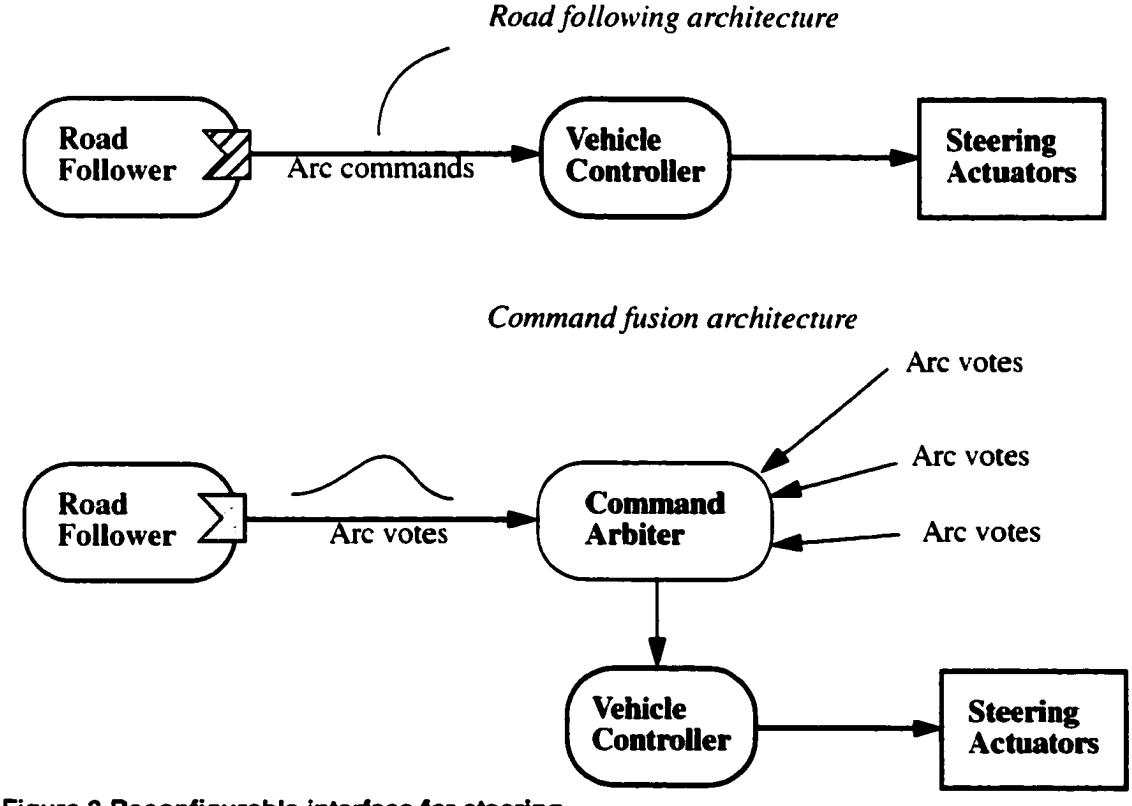


Figure 3 Reconfigurable interface for steering

1.7 Architectures and tasks

A priori architectures are still a part of the emergent architecture approach. An architecture developed at the beginning of a project, in the design phase, represents the best guess at satisfying the task requirements. In an emergent architecture, as the task requirements become apparent through actual experimentation, the system architecture is expected to change, sometimes in an incremental fashion and sometimes in a wholesale fashion. The expectation for change is built into the approach.

Existing architectures can also provide starting points and lessons that can be incorporated into a task's architecture. Chapter 3 presents some carefully chosen architectural schemes and explores what kinds of tasks are appropriate for each. These schemes are extremes on continua of possible architectures, and appropriate architectures for most tasks lie in the pragmatic intervals between the extremes.

Chapter 3 goes on to introduce my primary benchmark task, an integrated, multi-modal cross country navigation system. Finally Chapter 3 also presents my secondary benchmark task, an on road navigation system. There are some key differences in needs of the two tasks, even though they might be classified as belonging to the same domain. These differences in task requirements translate into differences in system architectures.

1.8 The case study: architectures for robotic ground vehicles

I devote the remainder of the thesis to applying the philosophy of emergent architectures to the specific benchmark tasks. Chapter 4 starts the process by choosing an architectural infrastructure, i.e., the means through which most modules in the system communicate with each other. The infrastructure is not the content of the communications, i.e., the protocols of interaction, but rather the medium through which modules communicate. This level of architectural description might be considered as too implementation specific to be important, but the implementation specific details of an architecture can determine whether it is effective or not.

Chapter 5 explores the current software architectures for the benchmark tasks. These architectures have emerged from the interaction of our architectural toolkits with the requirements of our

organization, hardware, and, most importantly, tasks. Chapter 5 goes on to show that the task requirements of my secondary benchmark task, the on-road navigation system, differ from the requirements of the primary benchmark task sufficiently that a different architecture is required to meet the new demands. While the architecture of the primary task might have "sufficed" for the secondary task, I argue that a different architecture is a better fit and results in a more capable system.

Chapter 5 also demonstrates how the emergent architecture toolkits provide the luxury of maintaining multiple architectures for multiple tasks while easily sharing and reusing code. Moreover, the chapter demonstrates how, as a side effect, the reconfigurable interfaces ease the debugging of modules within the same project by making specialized "debugging" architectures simple to construct and easy to use.

Every element of an emergent architecture must be in line with the requirements of the task and capabilities of the system. In my benchmark tasks the vast majority of the computational, developmental, and logistical resources are taken up by perception. Chapter 6 focuses specifically on how I do plan specification, execution, and monitoring in such perception centered systems. I present a plan execution system, SAUSAGES (System for AUtonomous Specification, Acquisition, Generation, and Execution of Schemata) that satisfies the current system requirements for a planning system and has the potential for satisfying future requirements. Chapter 7 will then show the benefits of using geometry in order to have plans that our robot system can actually execute in the real world rather than just in simulation.

It could be argued that the approach of emergent architectures only really fits a particular task with particular personnel. Chapter 8 will argue that there are a wide range of domains and situations in which the emergent architecture approach will be of great benefit, and it identifies some of the characteristics of such domains and situations.

Chapter 2 Reconfigurable Interfaces

2.1 Software reuse and reconfigurable interfaces

Developers of mobile robot systems have for the most part ignored the advice and lessons of software engineers for software reuse. The result is that most mobile robot systems are developed from scratch for one project that has one purpose. The one-of-a-kind approach makes moving software components, i.e., the individual modules and algorithms, from one project to another extremely difficult. Each module is crafted for the particular system in which it is embedded, and moving the module from system to system means going into its source code and making extensive changes. These changes have to be made in each module that is to be ported, and worse, the changes are usually different in each module that has to be ported.

The reason mobile robot engineers have ignored software engineers is simple: The reuse schemes of software engineers depend on early standardization of systems. Unfortunately, mobile robotics is not at a stage at which standardization is useful or appropriate. The hardware, algorithms, and ambitions are changing fast enough that standards specific enough to be useful quickly become obsolete.

In this chapter I present a set of tools for building practical, mobile robot systems with reusable components. I show how to use a reconfigurable interface system to separate the work of the algorithm developers from the work of the system developers. This lets system developers build

systems which can reuse software from other projects while still having a system that satisfies the needs of the current task.

2.2 Perspectives on software reuse

Researchers in software reuse have identified four areas that a reuse system should address[Biggerstaff and Perlis, 1989]:

- Finding components
- Understanding components
- Modifying components
- Composing components

The focus of many researchers in software reuse has been the total automation of this process, i.e., given a specification of the desired application, automatically choose and compose all of the components necessary to build the application. Total automation is not the goal of this thesis. In my work there is a human system integrator who shepherds the system building process for a system that performs a particular task, using some components that have been built previously and some that will be specially developed. My goal is to make the jobs of this integrator easier.

Many researchers concentrate on the first two elements of component reuse. The scenario that they envision is that of a library containing thousands of "widgets." The hard and interesting problems in this scenario involve retrieving the appropriate widget given a component specification. This scenario is almost totally irrelevant to the problems I am trying to solve. At this point in the field of mobile robots there are not thousands of widgets available to solve the problems. A system integrator will be lucky to have one component that solves any problem in a mobile robot system, and will certainly not have more than two or three. In the scenario that I envision, a mobile robot system integrator is given a component and must make it fit in more than one application. Eventually the "library problem" will be applicable, but it is not the place to start with this infant domain.

2.2.1 Bottom up reuse: Object oriented approaches

One approach to software reuse has been to come up with a standardized set of reusable components which can be put together to form many different systems. This approach is most successful in extremely formalized domains such as numerical analysis in which over five hundred years of experience serves as a rich source for useful standards. The more experience that exists with a field, the more robust the standards are, and the more applicable the reusable components can be.

One popular method of building reusable components that can be joined together in a bottom up fashion to build systems is the object oriented approach. Bhaskar warns that the phrase object oriented "has been bandied about with carefree abandon with much the same reverence accorded 'motherhood,' 'apple pie,' and 'structured programming'."[Bhaskar, 1983] Nevertheless, the definition of "object oriented" rests on the definition of an object. An object, according to Booch [Booch, 1991], has several major identifying properties:

- *Abstraction*. An object should have a description of its behavior that is available to outside entities. This description, or abstraction, should be a complete specification of how the object interacts with the world.
- *Encapsulation*. No outside entity should be aware of anything about the object except its abstraction, and all implementation specific information should be hidden, or encapsulated, inside the object.
- *Hierarchy*. Each object is an instance of a class, and the classes are organized in hierarchies. The class hierarchies give a means of organizing all of the different abstractions in a system.

An object oriented system is simply a set of objects operating on each other to perform some higher level task.[Booch, 1991] Exactly what objects with what class hierarchies make up an object oriented system is the result of object oriented decomposition, analysis, and design for the particular problem.

Figure 4 shows an object oriented system that solves the problem of following a road. In this simplified system there are three objects: an image source which provides pictures of the road in front of the vehicle, a road follower object which requests images from the image source and pro-

cesses them to figure out where the road is and where to steer, and a controller object which executes arcs given to it by other objects. Each of the objects is an instance of a class which is part of an "*is-a*" hierarchy, i.e., a **RoadFollower** object is a **Sensorimotor** object, which is a **Module**. The *is-a* relationship means that objects inherit the operations and methods of the classes above it in the *is-a* hierarchy. For example, the fact that everything *is-a* **Module** means that there are operations and interactions standard for all objects in the system, such as a common reset or stop method.

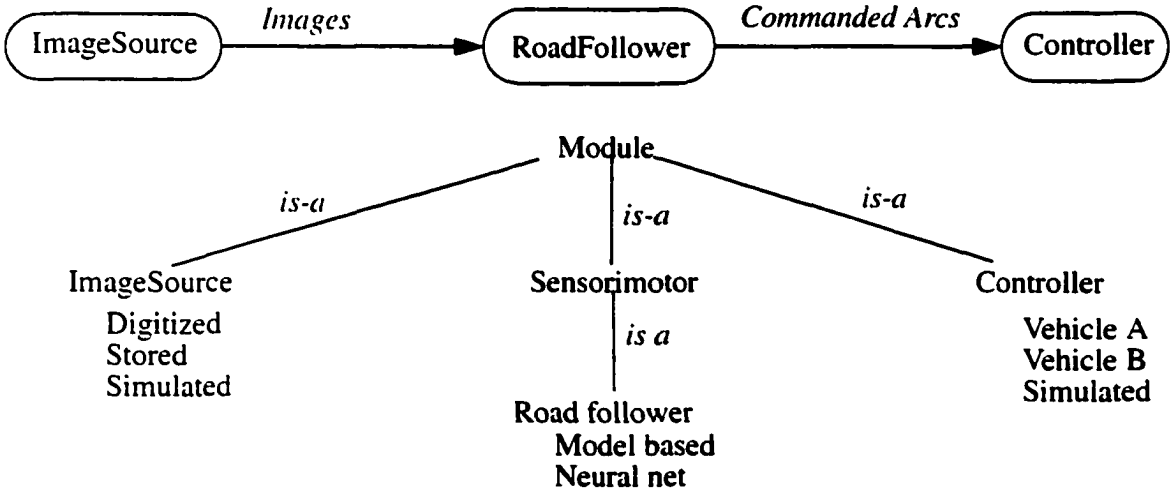


Figure 4 Object oriented decomposition of a road following system

The encapsulation of an object's implementation behind a barrier of abstraction provides some simple opportunities for reuse. The designer is free to build a system that uses live digitized data, stored video data, or simulated video data just by changing the implementation of **ImageSource**. Similarly, the system can be moved to various vehicles, or even run on a simulated vehicle simply by changing the implementation of the **Controller** object that is used.

The real power of the object oriented approach though comes from the power of inheritance and the class hierarchy. First, it provides a mechanism for polymorphism, i.e., having different implementations for the same abstraction. For example, rather than going into **ImageSource** and changing the implementation from using live digitized data to using simulated data, a system designer can create two subclasses of **ImageSource**, **DigitizedImageSource** and **SimulatedImageSource**, which each have different implementations of the abstraction specified by **Image-**

Source. A user of an **ImageSource** instance will not care which subclass the instance really is, as long as the subclass obeys the original abstraction.

A bigger advantage for code reuse that the subclassing and inheritance mechanisms of an object oriented system provide is the ability to incrementally add to and change a class's abstraction. For example, say that a **Controller** class has an operator "*drive*" which takes a turning curvature and speed and moves the steering wheel and brake/throttle actuators to the appropriate positions. In later development there might be a need for a drive operator which also takes bounds on the acceleration permitted to achieve the desired speed. Rather than rework every component that has ever used instances of **Controller**s, the system designers can create a subclass of **Controller**, **BoundedController**, which inherits the original *drive* operation and adds a *drive_bounded* operation. For a **BoundedController** a *drive* operation could be implemented using a *drive_bounded* operation with some default acceleration bounds. Thus, components in the system could still access the controller through the **Controller** abstraction without change, while components that need the extra operations can use the more specific abstraction defined by the **BoundedController** subclass.

The basis for an object oriented system is a good decomposition of the problem into classes. This decomposition usually represents a solution to a particular problem at a particular time. The decomposition can be incrementally changed over time using the powers of inheritance within the class hierarchy, and the individual components could be used in a variety of systems for which the decomposition holds.

Unfortunately, that decomposition may not be the best for all related problems, for a different research group working on the same problem, or even for the same problem in a few years. Research with human subjects has shown that "there are potentially at least as many ways of dividing up the world into object systems as there are scientists to undertake the task."[Coombs et al., 1954]

For example, Figure 5 shows a different decomposition for a road following system. In this decomposition the road follower object outputs semantic information about the world, i.e., where the road is, to a knowledge integrator. The knowledge integrator object also supplies the road follower with the appropriate images and outputs actuator commands to a controller object. In this

different decomposition, the controller and image source object could use the same class definitions as previously, but there is no way that a road follower object could be defined in the same way: In this decomposition of the problem the road follower interacts with completely different classes and abstractions. The assumptions driving this decomposition are completely different from the previous decomposition, and the results are modules that cannot be simply "moved" from system to system using the standard object model. The algorithm for road following may be the same, how the algorithm gets its inputs and where it puts its outputs is married to the particular object oriented decomposition of the system, e.g., in the system of Figure 5 the road follower object must contact a knowledge integrator object and request images and deposit road information while in the system of Figure 4 the road follower must contact an image source object to acquire an image and contact a controller object to output where to steer the vehicle.

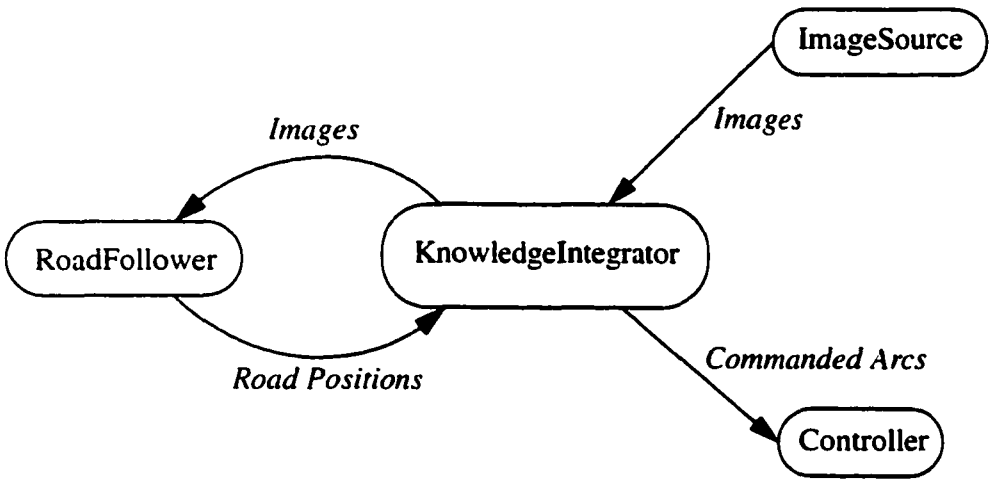


Figure 5 A different approach to road following

So, a given class hierarchy for a mobile robot system can only represent a snapshot of a given solution to a given problem by a given set of researchers. If any of these variables change, the "best" classification changes as well. True "domain" standards arise from extensive experience with the domain and experimentation with what works best for given problems. There has not been enough experience with mobile robots interacting with real problems to decompose the mobile robot "problem" into a set of classified objects which can be glued together for particular applications.

2.2.2 Top down reuse: Architectural approach

The object model is one way of building standardized components that can be built up to make various systems. An alternative is a "top down" approach, in which what is standardized and reused is the topology and pattern of interactions between components rather than the components alone, i.e., create a standard software architecture and use instantiations of that architecture to build different systems. Of course, there cannot be any one architecture that satisfies all needs, but the argument is that there can be standardized architectures specific to single domains [Mettala and Graham, 1992]. The most important part of an architecture based approach is a *reference architecture* that specifies the overall structure of how components communicate. Since any task is the result of a variant of the reference architecture, a system integrator can categorize every module in the system into a slot in the architecture, and thus modules can transition easily from task to task within the domain since they regardless of the architectural variant they are in, they remain in the same "slot." Another important component is the *component library*, which contains task specific code and the methods by which individual modules actually communicate. The dream of the architecture based efforts such as the Domain-Specific Software Architecture (DSSA) program[Hayes-Roth, 1994] is to take a task specification for a domain, and then automatically select the best components and slight modifications to the reference architecture to present an executable program which does the job.

The architectural approach has a fundamental conflict as far as a young, unstable field such as mobile robots is concerned. In order for a module to be reusable it needs to know about the reference architecture. The design of the module depends on the design of the reference architecture because otherwise the module would not be transferrable from task to task within the domain that the reference architecture implements. The conflict is between the scope of the domain and the applicability of the domain. If the domain is too broad so that many tasks can fall under its umbrella, how do we know if that domain's reference architecture is truly applicable to all of its possible tasks? On the other hand, if the domain is so narrow that we can predict closely what all the possible task requirements are, we lose the ability to easily reuse modules in tasks that are related but not quite in the same domain.

Whatever the scope or applicability of a reference architecture, there is still a basic problem: What if contact with the "real world" shows the reference architecture to be flawed? If every module in the system reflects the structure of the reference architecture, a change in the architecture can require re-engineering every module in the system. This kind of re-engineering directly conflicts with the main goal of the architectural approach, that is easing software reuse.

The top down approaches assume that a domain is stable and well known enough to come with a single reference architecture, and all problems can then be expressed as slight variations of this reference architecture. Unfortunately, the "domain" of mobile robots is neither explored nor stable, nor even sufficiently defined, to even declare it as a single domain, let alone come up with the reference architecture. Attempts to define a reference architecture at this point are futile, and will not encourage code reuse.

2.3 Reconfigurable interfaces

To what extent is code reuse possible in a field for which the capabilities and requirements are constantly changing? I claim that even though the overall topology of a system and the methods for how subsystems communicate may be changing, it is still possible and necessary to achieve significant levels of component level reuse. The means I propose is reconfigurable interfaces.

Reconfigurable interfaces provide a mechanism by which an individual module in a mobile robot system interacts with abstract data sources and destinations without regard to how those sources and destinations are implemented. These abstract sources and destinations act as adaptors or mediators written by system integrators to stand between the module and the current instance of the system software architecture. For example, a module might need to know the position of the robot, so it would access a "position source." In early development, the positions might simply be read from a file, thus allowing the module to be run all by itself. In later stages of development positions might come from an external vehicle controller process, entailing a whole runtime structure of other modules to be brought up and used via an interprocess communication system. Similarly, a module might produce an arc on which to drive the vehicle. In the initial standalone testing this might simply be printed to the screen, while in later integrated testing the arc might have to be converted into a set of arc "votes" and sent to an external arbiter process.

The implementation of the sources and destinations is software architecture dependent. They depend on the particular architectural infrastructure and topology of the software system. Reconfigurable interfaces ease the task of moving a module from architecture to architecture, and allow more freedom in modifying the software architecture of a system as new requirements are discovered and new capabilities implemented.

Reconfigurable interfaces allow individual modules in the system to have a "module centric" view of the system. As far as the individual modules are concerned, they are the center of the system. Whether this is true or not should be hidden from the module and the module developer. This approach turns the normal top-down design process on its head. In a normal top-down design process a design team will decide that they need a module "foo" to perform some purpose, and the module programmer writes and uses code to fit the module into its prescribed slot in the system. Reconfigurable interfaces free a module to choose the interfaces it will use based on what is available, convenient, and appropriate, not based on fitting into any particular run time structure. It is the job of the system integrators to direct and transform the modules' inputs and outputs within the context of a particular software architecture, not the job of the module developers.

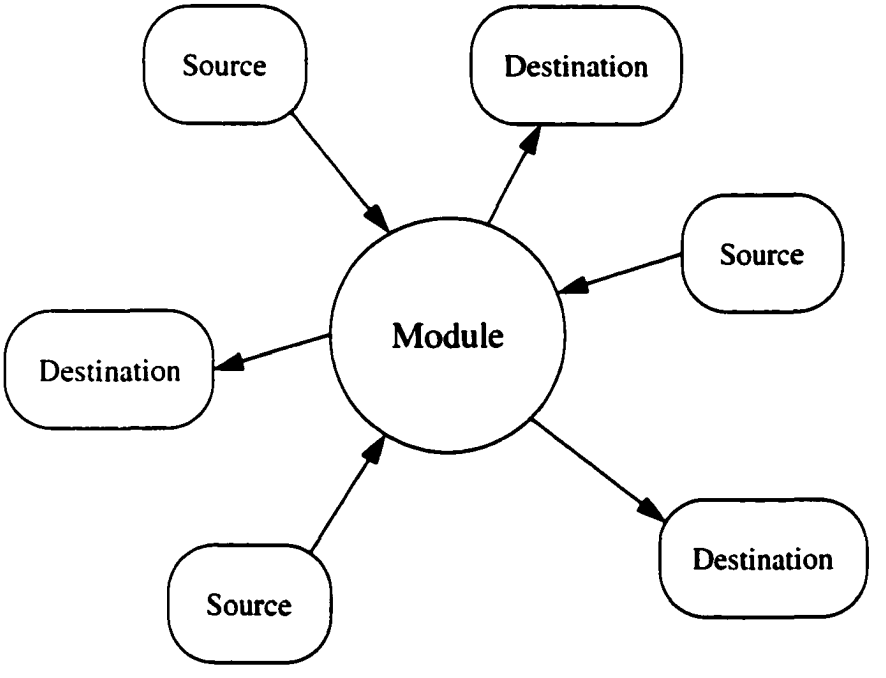


Figure 6 A module's view of the software architecture

Reconfigurable interfaces free system designers to take "components" from other projects with different software architectures, write "adapters" and plug the component into the new system without enforcing broad, permanent standards on every worker in a field which is not ready for standards. Thus, there can be different software architectures for different tasks that share components.

Reconfigurable interfaces also address the fact that architectures don't just change from project to project, they change from day to day. Figure 7 shows the daily architectures in the development of a sensorimotor perception subsystem **P**. For most of the development and debugging of **P** the best "architecture" is stand-alone passive, i.e., the inputs to **P** are simulated data, either from simulators or from data taken in a real run. This architecture provides the most controlled input for debugging and development with the least amount of architectural overhead. The output at this stage is used purely for debugging and verification. Once as much of the development and debugging has been done using the simulated and canned input, the developer will take **P** and put it on the vehicle. The developer will run **P** in stand-alone active mode, using real data from real sensors and sending real actuator commands. The developer uses the stand-alone active stage to debug all of the unforeseen interactions between the module and the real world. The final stage is the integrated system, in which **P** is controlled by some external modules and its outputs are merged with other perception modules to perform more complex tasks than it could perform alone.

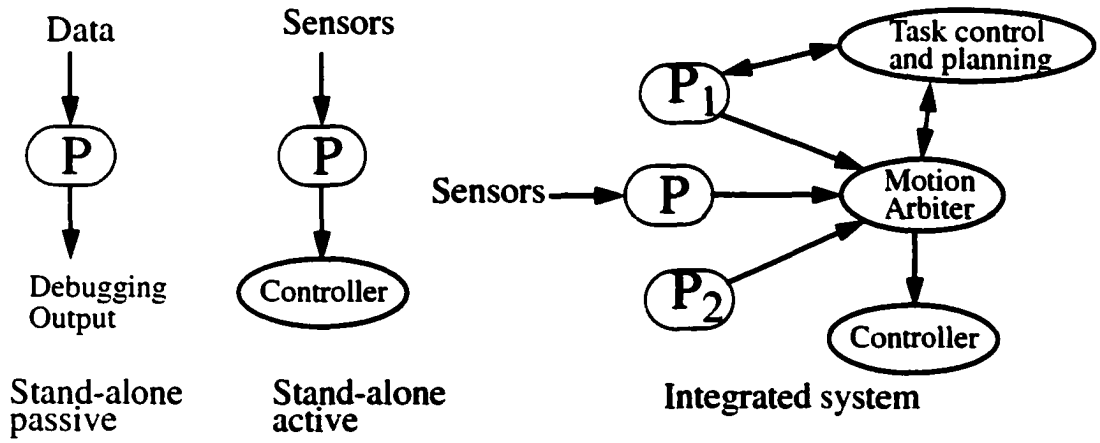
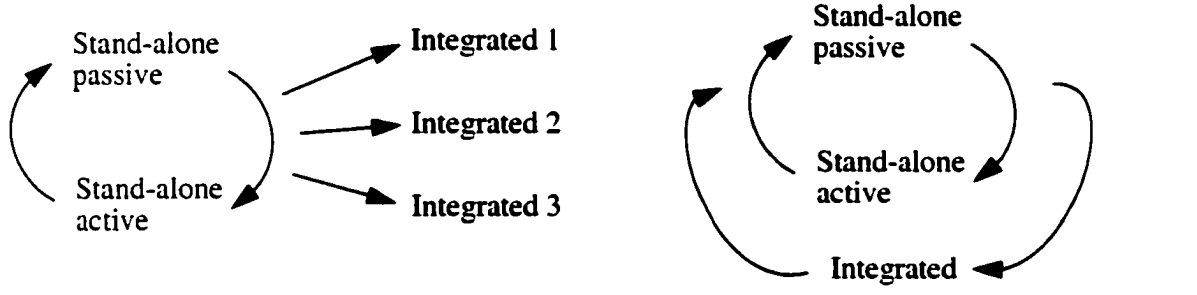


Figure 7 The daily architectures for the development of a perception subsystem

Figure 8a shows the typical evolution of a perception subsystem that does not use reconfigurable interfaces. Most of the development time is spent in a cycle of lab testing with simulated and canned data and stand-alone testing with real data. When the algorithm is stable, then the module developer works with the system developer to create a version of the code that can be integrated into a larger system. Unfortunately, the code for controlling the algorithm, getting at system resources, and outputting the results often has to be changed. Because the module developer is primarily concerned with his stand-alone testing, often what results is a separate integrated version of the module. The module developer hands this integrated version off to the system developer, who tests it interacting with the rest of the system. The problem is that then the module developer returns to the original development cycle. Every time there comes a time to "integrate" the algorithm, the module developer and the system developer must collaborate again to create another version to integrate into the larger architecture. Even if the module developer has developed some scheme for isolating the algorithm from the architecture there will still come breakdowns when the architecture changes: A change to the integrated architecture means going in and changing code to make the algorithm fit into the new scheme.

Figure 8b shows the goal of reconfigurable interfaces: a continuous development cycle. In this process there are no discrete "integration" points, where a system developer takes the algorithm and "integrates" it. Integration should be seamless and immediate upon reconfiguring the interfaces. The isolation of interface from algorithm lets a module developer develop and perfect the algorithm while a system developer develops the architecture, allowing each of them to concurrently do the job for which they are best qualified.



a) Fragmented development cycle

b) Continuous development cycle

Figure 8 Two possible development cycles

2.3.1 Mediation and negotiation

The concept of isolating components from the architecture in which they reside is not new. The reconfigurable interfaces approach is very similar to the "mediated" approach to information management[Huhns and Singh, 1995, Wiederhold, 1992]. In such a system all knowledge bases, applications, interfaces, and databases view the rest of the system through "mediators." These mediators are custom designed by system integrators to isolate the individual components from the system as a whole, so as to translate and route information and requests generated by the component into formats that are meaningful for the system as a whole.

The goals of the mediator architecture and of my reconfigurable interfaces are scalability (maximally ease addition to the system), maintainability (maximally ease change to the system), and efficiency (minimize the impact on component interoperation). Unfortunately, in the mediator architecture these are listed in decreasing order of importance while in the domains of the reconfigurable interfaces these are listed in increasing order of importance. The mediator architecture is primarily aimed at the integration and interoperation of very different kinds of commercial, off-the-shelf data bases. In this domain efficiency is an issue, but the success or failure of a system is fundamentally determined by its scalability and maintainability, not its efficiency. In the mobile robot domain, efficiency of interoperation has a much greater impact on the success or failure of a system, and thus the reconfigurable interfaces can be seen as a variant of the mediator architecture which takes into account this fundamental domain difference.

Both the mediator approach and the reconfigurable interfaces approach use interfaces generated and tuned by human system integrators to regulate the interaction between a component and the system in which it resides. An alternative is to use an agent-based approach in which the knowledge that a component requires and produces is described using a semantic meta-language such as KQML[Finin et al., 1994]. Each component, which could be better considered as an agent, would then "negotiate" for what it needs from the system, i.e., it would find the sources of the knowledge it needs, find out what format that knowledge is in, and translate it as necessary. Similarly each agent component would advertize the knowledge it produces in a way that other agent components could acquire it. The system integrator would provide facilities for locating

knowledge sources and destination, for reasoning about knowledge, and for converting knowledge between different representations.

If a software system is highly dynamic at run-time, i.e., there is a very high probability of qualitative system changes happening during the run of the system, then considering components as agents negotiating for knowledge is appropriate. The mobile robot systems I consider here are static once the system starts up, so using a negotiating agent model of interfacing would be overkill, i.e., the result would be a degenerate case of the negotiating agent model. In the degenerate case the system would incur all of the overhead problems without needing the flexibility benefits. In addition, the role of the system integrator is changed from fine-tuning interfaces to fine-tuning reasoning and conversion procedures. While ultimately this could result in making the system integrator's job easier, it will reduce the level of immediate control the system integrator has over exactly how two given components will interact.

For some tasks these problems may not weigh as heavily, and in which case a negotiation based approach to reconfigurable interfaces may be appropriate. One way to look at this would be that for those systems, the reconfigurable interfaces are simply trivial "covers" for the underlying agent negotiation scheme. Thus we may ease the moving of a module that exists in such a flexible architectural environment to a more constrained architectural environment, since the transition will be through making the reconfigurable interface implementations non-trivial rather than forcing the module to be reprogrammed to deal with a non-trivial interface. In essence, I suggest that for some class of tasks a negotiation, or agent, based approach may be an appropriate architectural infrastructure rather than being an appropriate approach to reconfigurable interfaces as a whole (see Section 4.3.4 for more on this).

2.3.2 Requirements

In order to be useful in mobile robots, reconfigurable interfaces cannot be allowed to cripple the functionality of the system. The tasks in mobile robotics run close to the edge in terms of performance, and developers will not use a reconfigurable interface system if it bogs down the system performance. The overhead incurred by using reconfigurable interfaces need to be kept within acceptable limits, with the limits being defined by the particular task.

Despite these considerable constraints, interfaces that can be reconfigured at start up time, i.e., not necessarily at run time, but reconfigurable without recompilation, are still necessary. The reason can be seen in the "daily architectures" cycle. Switching a module from architecture to architecture is not just something to be done when the module is ported from project to project, but it should be happening on an almost daily level. For example, take a system that is being tested, and say that a bug is found in module A. We want to take out module A, test it on it own, possibly with data collected during the integrated run, fix the bug, and then reintegrate it. This whole cycle might happen over the course of a few hours. Not having to recompile to move the module from developmental stage to developmental stage will increase the efficiency of isolating and fixing bugs that occur in large integrated systems.

An overarching concern is the impact of the reconfigurable interfaces on the module programmers. I do not have dictatorial powers over the module developers, and even if I did, if I forced the philosophy of reconfigurable interfaces on them, if it is a burden to them, they will find ways to avoid it. The reconfigurable interfaces must be easy for the module developers to use. In fact, a goal of the reconfigurable interfaces should be to add functionality that module developers need over the existing interfaces. The more useful the reconfigurable interfaces are for developers other than the system integrator, the more likely it will be that module developers will embrace the philosophy of reconfigurable interfaces. More module developers embracing the philosophy of reconfigurable interfaces will lead, in the long run, to easier system integration and development, so any effort the system integrator puts into making the reconfigurable interfaces useful rather than burdensome will pay off in the end.

One more requirement flows from the fact that the system developer does not have dictatorial control over the module programmers: The reconfigurable interface system should be able to stand on its own on a module by module basis. There is no guarantee that all module programmers will adopt the reconfigurable interface philosophy. This can be a problem if the implementation of the reconfigurable interfaces depends on the fact that everyone involved in developing the system and components of the system uses them. This problem must be faced, especially when introducing the reconfigurable interface philosophy part way through the development of a system. The goal of a reconfigurable interface system is to be something that can take over a project by being useful for development as well as integration. It should not be the case that in order to

use the reconfigurable interfaces all components in the system must be changed at once. This factor becomes even more important when moving code from one system at one site to another system at another site. When porting the code, I cannot demand that the other site switch over to my philosophy. A goal of reconfigurable interfaces should be to make this move easier, not harder.

If a reconfigurable interface system is incremental and attractive to developers rather than system wide and burdensome to developers it solves two of the major obstacles that researchers have identified in the adoption of software reuse schemes: capital investment and the Not Invented Here (NIH) syndrome. If adopting a reuse scheme entails changing every component that has been developed at a site, the initial capital investment in terms of the time and effort needed will be daunting to accept, even with the long term benefits of software reuse. The NIH syndrome is especially rampant in a research and development environment, in which it can become the Not Invented By Me syndrome. The NIH syndrome results from developers not believing that an externally imposed, system wide scheme is completely applicable to their particular needs. The NIH syndrome can be overcome if the reuse scheme does not hurt the applications efficiency, and provides immediate, certain benefits in development and debugging, rather than long term, uncertain benefits for code reuse.

2.3.3 Options

Isolating the algorithm from the interface is something that most good module developers do instinctively. Most of them recognize that their systems go through the development cycles of the daily architectures indicated in Figure 7. Unfortunately, each of them uses their own approach to reconfigurable interfaces, and the approaches are usually limited to only the architectures that they know their systems are involved in. A change in the architecture, or porting the code from one architecture to another, involves going into each module programmers code and changing it, with different changes required for different modules.

2.3.3.1 Compile time: #ifdefs

One homespun approach to reconfigurable interfaces is to include sections of code that compile differently under different "architectures." In C, this is done with the **#ifdef** construct. For example, in the following code the module programmer specifies that the system should do one

thing under architecture FOO and another under architecture BAR. This is resolved at compile time and done through compiler switches.

```
#ifdef FOO
     do_one_thing();
#ifdef BAR
     do_another();
```

This approach is not often used to address the "daily architectures" problem. Most of the time it is used when the same module is used in two different projects with two different architectures, allowing the module developer to maintain code consistency between the two projects.

There are many problems with this approach. First, it severely degrades the readability and maintainability of the code. Second, only a fixed number of architectural reconfigurations are allowed. If the code is to be ported to yet another architecture, that is yet another set of #ifdef's that need to be added. If the architectures change, then the code itself has to be changed as well. Finally, there is no guarantee that each module programmer will use the same set of **#ifdef**'s, with the resulting chaos making the job of the system integrator much harder as the system develops and changes.

2.3.3.2 Run time: switch/case statements

A common approach to the reconfiguring for the daily architectures, i.e., switching a module between stand alone passive, stand alone active, and stand alone integrated testing, is through sets of conditional statements. In C or C++ this is usually done with large **switch/case** statements to differentiate at run time between architectures.

```
switch (architecture_type) {
     case FOO:
          do_one_thing();
          break;
     case BAR:
          do_another();
          break;
}
```

This approach is good for switching between the daily architectures at run time, but it still limits the choices of architectures to a fixed set. If the system is moved to a completely different architecture or the architecture changes radically then the module code needs to be changed again.

2.3.3.3 Link time: encapsulating libraries

One of the best approaches I have seen for isolating modules from the architecture is encapsulating libraries. The concept is simple: maintain a library with a consistent interface but different implementations, one for each architecture that is relevant. In order to switch from architecture to architecture, simply link with a different library.

Figure 9 Encapsulating libraries as reconfigurable interfaces

A more concrete example is a library that returns a two dimensional position. The single call in the library would be

```
int posGet(float* x, float* y, float* heading);
```

Different libraries would implement this function differently. On a shared memory, real time system, this would be implemented as the reading of a shared memory location. On a UNIX based system the implementation might be a TCP/IP query to an external server. In simulation the implementation reads the position from a file. The library encapsulates all of these architectural issues, thus isolating the module from the run time configuration and architectural infrastructure.

Unlike the previous methods, the encapsulating libraries approach does not suffer from a limited set of options. System developers are free to write new libraries, and a module can immediately move from architecture to architecture just by switching which libraries are in use. What this method lacks is run-time flexibility. In order to move from architecture to architecture modules must be relinked. This makes pulling out a module from a system, testing it, and putting it back in the system more of a chore. The approach also lacks a consistent method for configuring the implementations. For example, in previous example, the simulated position source will need to be configured with the name of the position file while the shared memory implementation will need to be configured with the address of the 2D position information. There is no easy way to pass in this configuration information in a straight forward implementation of the encapsulating libraries approach.

2.4 Emergent architecture toolkit

Of the potential bases for a reconfigurable interface system, the one that comes closest is the encapsulated libraries approach. An encapsulating library provides a simple, standardized way for an end user to interact with the resources an architecture provides without much overhead at run time. The cost, of course, is flexibility. Whatever the library is configured for at link time, that is what the end users get until they relink.

I achieve the necessary level of start-up time flexibility through the idea of switchable encapsulated libraries. The concept is simple: rather than having one library at a time resident in memory, there are several. As with the standard encapsulated library approach, each of these libraries has the same interface, but the user can switch between them at run time using a specification string sent to an initialization routine. The specification string not only selects the library, it also passes in parameters to be used by the library for initialization.

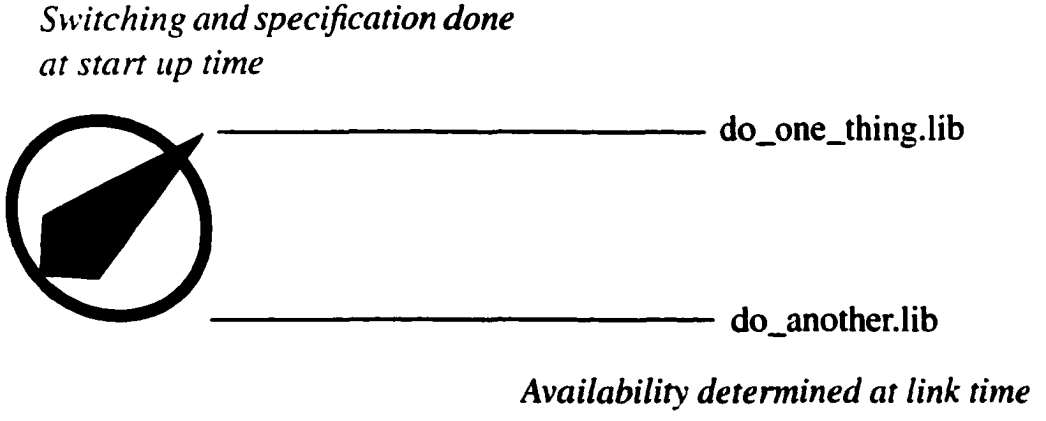


Figure 10 Start-up time switching between libraries

The library encapsulates many of the architectural issues, so the choice of the library determines which architecture the module will interact with. This switching means that we can make the libraries for the "daily" architectures available at the same time and allow users to switch between them at run-time without any recompilation. This satisfies the run time flexibility requirement and makes debugging of modules easier due to the reduction in time and effort of moving a module from simulation to stand alone testing, to real world testing in an integrated system.

2.4.1 Example: A position/state source interface

The example I will use in explaining the implementation of my reconfigurable interfaces is a position/state source interface. A position/state source supplies information about the local 2D and 3D position of the vehicle, the global 2D position of the vehicle, and the state of the vehicle, i.e., how fast the vehicle is moving, the turn radius of the steering wheel, and the distance traveled since initialization. This position/state source interface is actually used in our systems and provides a consistent method of accessing vehicle information. To a C programmer, the interface looks like this,

```
int pssPos2D(PosStateSource* source,
             double* x, double* y, double* heading);
int pssPos3D(PosStateSource* source,
             double* x, double* y, double* z,
             double* roll, double* pitch, double* yaw);
int pssPosGlobal(PosStateSource* source,
                 double* x, double* y, double* heading);
int pssState(PosStateSource* source,
             double* speed, double* curvature, double* dist);
int pssPosState(PosStateSource* source,
             double* x, double* y, double* th,
             double* speed, double* curv, double* dist);
double pssCurvature(PosStateSource* source);
double pssSpeed(PosStateSource* source);
double pssDistTraveled(PosStateSource* source);
```

All of the interface routines take a position/state source interface instance as the first argument. This interface instance contains all of the necessary information to get the position and state of the vehicle from a given source. Most of the interface routines return an integer status. Some of the interface routines could conceivably be built out of the others. For example, **pssCurvature** could be implemented using **pssState** to find the current steering curvature and return it. Whether that is the case will vary from instance to instance.

2.4.2 Creating the interface instance

There are two parts to creating an interface instance. First of all, there will be many interfaces, and many of them will interact in ways that will be hidden from the end user. For example, both a position/state source interface instance and an actuator interface instance (i.e., something that sets the steering curvature and speed) may have to communicate to the same controller which actually moves the actuators and monitors their position. In order to facilitate these interactions, the user

of the reconfigurable interfaces must provide a pool of "globals" to the instantiation commands. This pool of globals is simply a table of name/value pairs. The values contained in the globals pool can be parameters which will be common to many interface instances or structures by which interface instances will interact, such as controller interfaces.

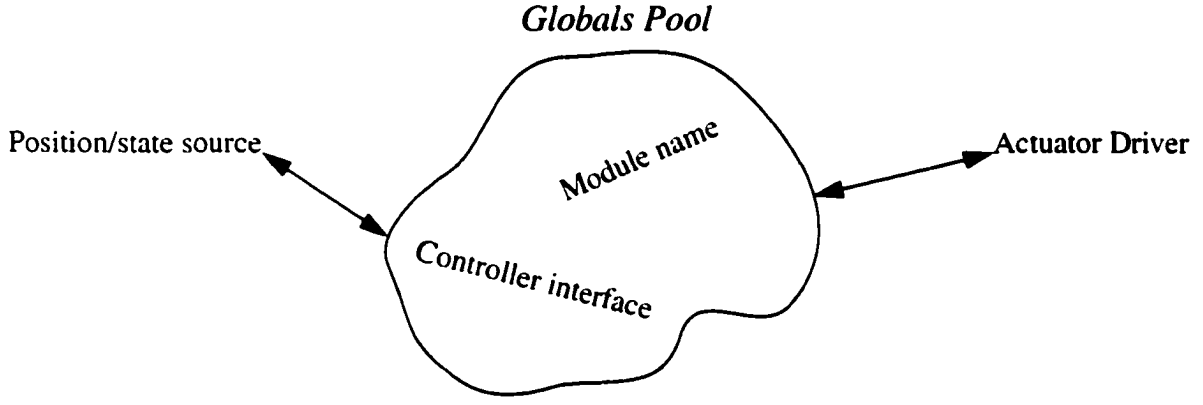


Figure 11 Interface instances interacting through the globals pool

To actually create an interface instance, the user passes the globals pool and a specification string to a creation function. The specification string specifies which class of interface instance to instantiate and what the parameters are for the instantiation. Take the example of a module which is only going to need a position/state source. To initialize the position state source, a C programmer would write.

```
AOSymTable* globals;
PosStateSource* pss;
char* spec_string;

globals = symTableCreate(); /* create the globals pool */

/*   the parameter module_name is common to many interface
     instances, set it to "foo" */
symTableAdd(globals, "module_name", "foo");

/* create the position/state source instance */
pss = pssCreate(spec_string, globals);
```

The specification string should be initialized at run time, either through an initialization file, an environment variable, or even a command line argument. Only through run time initialization of the specification string can the user take advantage of the abilities to change architectures at run time that the reconfigurable interfaces provide.

2.4.3 Specification string syntax

A specification string always contains a "tag," or a word that specifies which encapsulated library to use. If no library with that tag is available, then the creation routine will return a null value. A specification string can consist of just a tag, which means that the interface instance will be created using all default parameters. For example, one position/state source we have implemented is an internal simulator. This position state source creates a vehicle simulator structure which maintains the vehicle position and moves the vehicle position as the vehicle simulator receives vehicle commands. This vehicle simulator structure is placed in the pool of globals to be used by other interfaces. To choose this interface instance, the user just passes in the string

```
"sim"
```

to the position/state source creation function.

The user can add parameters to the specification string. Say that the user wanted to specify a simulator with a given initial position and orientation. That specification string might be

```
"sim: initial_x=100.0; initial_y=100.0; initial_th=-1.5;"
```

The parameters given by the user in the specification string are transformed into a symbol table that the initializer for a interface instance receives. The initializer looks for parameters by name in the parameter symbol table and in the global symbol table, and uses default values for any parameters not defined.

2.4.4 Composite interfaces

As described so far, this approach to reconfigurable interfaces has only a small advantage in flexibility over the straight forward encapsulated libraries approach. The real power of the switchable approach is the ability to have composite interface instances. A composite interface is an interface instance that "contains" some other interface instance. When the user invokes a routine in a composite interface, that routine performs some operation as well as invoking the same routine of another interface instance.

For example, one composite interface provided for the position/state source libraries is the debug composite interface. This interface prints out debugging information about the calls being performed. For example, if the specification string is the following,

```
"debug: body=sim; log_file=log.out; "
```

then when the user invokes **pssGet2D**, the results, i.e., the 2 dimensional position, will be stored in the file log.out. Similarly, with the specification string,

```
"debug: body=(sim: initial_x=10.0; initial_y=15.0;) interval=1.0;"
```

the position and state routines will return their values gotten from an internal simulator initialized to x=10 and y=15, and those values will be printed to the screen at most every second.

Composite interface instances bring this approach to reconfigurable interfaces to real usefulness. They mean that the system is not just a switch between libraries, i.e., a boon to the system integrator, but that the system is of real benefit to the module developers as a debugging tool to see exactly what is going on. A good set of debugging composite interfaces can make the reconfigurable library approach extremely attractive to otherwise uninterested module developers.

Composite interface instances do not just have to be for debugging. For example, we have a module in our system that maintains a transform between the position reported by the vehicle controller and a position in the map. All a client has to do to subscribe to this position transform and apply it to get global positions is to use the "transformed" position state source composite instance. For example, the specification string

```
"transformed: body=cmu; "
```

will mean that the **pssGetGlobal** routine will return the local position transformed by the latest local to global position transform gotten from the transform maintenance module. The local position is gotten from the body of the composite, which is the interface instance which connects to the CMU controller.

2.4.5 Implementation

The reconfigurable interfaces are usually implemented in C++ because we can take advantage of the C++'s class hierarchies and inheritance schemes to reduce the amount of code that is reim-

plemented between different interface instances. For example, in the base class for the position/state source interface the pssPosState function is implemented as

```
int pssPosState(PosStateSource* source,
                double* x, double* y, double* th,
                double* speed, double* curv, double* dist) {
    if (pssPos2D(source, x, y, th) == FAILURE)
        return FAILURE;
    if pssState(source, speed, curvature, dist) == FAILURE)
        return FAILURE;
    return SUCCESS;
}
```

C++ allows the internally simulated position/state source subclass to inherit this default definition from the base class, while the interface to the real controller can take advantage of the fact that there is one command that requests position and state, and override the base definition of pssPosState, thus saving a interprocess query and response for getting position and state separately.

Of course, I cannot force all of the module developers to use C++, thus the interfaces have extensive support for C. C++ just provides a more natural framework for developing the interfaces since the class inheritance system provides a simple way of relating different interfaces in a way that allows code to be reused.

Each reconfigurable interface library has a configuration source file.The configuration source file includes code that installs pointers to interface creation functions in a hash table. When the specification string is parsed, the "tag," or interface name, is stripped out and the appropriate creation function is called with the rest of the specification string and a pointer to the globals pool. The creation function actually creates the reconfigurable interface instance, a C++ class, and returns it. Once the user instantiates a interface, actually invoking a reconfigurable interface method function only incurs the overhead necessary to invoke an inherited C++ function, which is usually no more than a pointer dereferencing.

2.4.6 Building reconfigurable interfaces

Actually building a reconfigurable interface is simple, if somewhat tedious at this point. I provide the user with a template program. Given the name of the reconfigurable interface, the template program creates the outline of a reconfigurable interface, with a skeleton configurator, and skeletons of the composite class, a debugging class, and a C expansion class as well as example

implementations of the C cover functions for the class. Using the templating program means that reconfigurable interfaces have a consistent look and basic implementation. The tedious part is that the developer has to go into the header files and implementations of all of the sample classes and declare and implement the member functions of the reconfigurable interfaces. Once the sample classes have all of the appropriate member functions, the developers can create their own interface subclasses and add them to the configuration routine so that the end user can access them.

There is much potential for further automating the tedious process of building a reconfigurable interface. I could take advantage of the Interface Description Languages (IDL's) used in some of the object oriented schemes to reduce the amount of repetitious implementation work that the user needs to do now to make using and building reconfigurable interfaces even more attractive.

An important temptation to avoid is to pack a reconfigurable interface with every possible interface class conceivable. An example from my experience is an interface to a digitizer/display. Over the history of a group there can be several digitizers and displays used. The temptation is to put interfaces for all of the digitizers ever used by the group into the reconfigurable interface. The problem is that most of the time there is only need for one digitizer interface. Each needless interface that is kept "active" represents space on disk and in memory that is not needed and time to load that is unnecessary. Thus, it is a good practice to keep the number of "active" classes in a library to the minimum required for that project at that time. If the older classes are needed it is simple enough to create another version of the same reconfigurable interface with those classes activated.

2.5 Where do reconfigurable interfaces come from?

Reconfigurable interfaces are primarily driven by what an individual module needs and the desire to provide for those needs and isolate the module from the means by which those needs are satisfied. The reconfigurable interfaces provides an abstract way to satisfy the modules concrete needs. For example, a stop sign detector will need a means of acquiring images. We need to provide a reconfigurable interface that lets the stop sign detector acquire and manipulate images in an efficient manner. At the same time we do not want to have to change the stop sign detection code

when we change digitizers, or if we use canned images stored in files, or if we use a system in which a separate module digitizes the image and sends the image to the stop sign detector. The developer of the stop sign detector only knows that there is an interface to an image source. The developer should not have to worry about what that image source is.

The absurd extreme of this approach to interfaces is that every module in the system has its own set of reconfigurable interfaces, each tailored exactly to the needs of that module. The resulting chaos of slightly different reconfigurable interfaces would result in a system as hard to maintain for a system integrator as if the reconfigurable interfaces were not used.

2.5.1 Commonalities and architectures

Fortunately, this absurd extreme is never reached in real system development. There are always commonalities and patterns of usage that allow useful reconfigurable interfaces to be shared by many modules in a system. For example, many modules in a mobile robot system will need an interface to an image source, so the reconfigurable image source interface can be shared by many of these vision based modules. Existing (and proposed) run time structures can also point to commonalities between modules that can be exploited through shared reconfigurable interfaces. For example, if we have a run time structure proposed in which several modules will be voting for steering directions, we can deduce that a single reconfigurable interface to an "arc voter" will be useful and reusable for many modules.

Most of the needs of individual modules in a system should be satisfied through the existing reconfigurable interfaces that have been tested thoroughly by the system integrators in the development and integration of other modules. There will still be the occasional need to develop a new reconfigurable interface. One reason this might happen is due to the inevitable trade-offs between generality and performance. If the system developers provide a general interface to an image source, there is the possibility that the very generality of the interface will incur overhead that the individual module developer finds unacceptable. In this case, a new interface which is less general but more efficient will have to be developed to satisfy the new needs of the system. Of course if one module developer needs the more efficient but less general interface, there is a good chance that others will need it as well, although that is not certainly true.

Another more common situation in which specialized reconfigurable interfaces will have to be built is for the output of a module. For example, the stop sign detector needs to do something with its results. Thus a reconfigurable interface for outputting the results of a stop sign detector needs to be built. In some run time structures the output will be text reports only, in another to a graphical display, and in another sent as a message to a semantic knowledge interpreter. The problem is that there will probably not be a high system wide demand for a reconfigurable interface to a stop sign detector destination. In all likelihood this will be a reconfigurable interface only used by one module, the stop sign detector.

2.5.2 Changing architectural paradigms

A reconfigurable interface should isolate the individual module from any changes in architectural paradigms. Figure 12 shows how a stop sign detector might be used for most of its development, in a run time structure in which external clients hook up specifically to the stop sign detector. In this run time structure it makes sense that there are reconfigurable interfaces for "stop sign destination" and "stop sign source." Eventually, there might come a architectural paradigm shift where there is a central geometric "token" collector/distributer, which takes geometric information about general objects in the world. Modules built within the context of this new run time structure would have reconfigurable interfaces for connecting to the general geometric token collector/distributer. The reconfigurable interfaces should protect the stop sign detector (and the stop sign clients) from knowing about this new paradigm. The "stop sign destination" interface would be reconfigured to send tokens to the geometric token center, and the "stop sign source" interfaces would be reconfigured to collect stop sign tokens from the token center. The system developers

are free to change paradigms, as long as they provide the "adaptors" or "mediators" for systems developed within the context of the old paradigms.

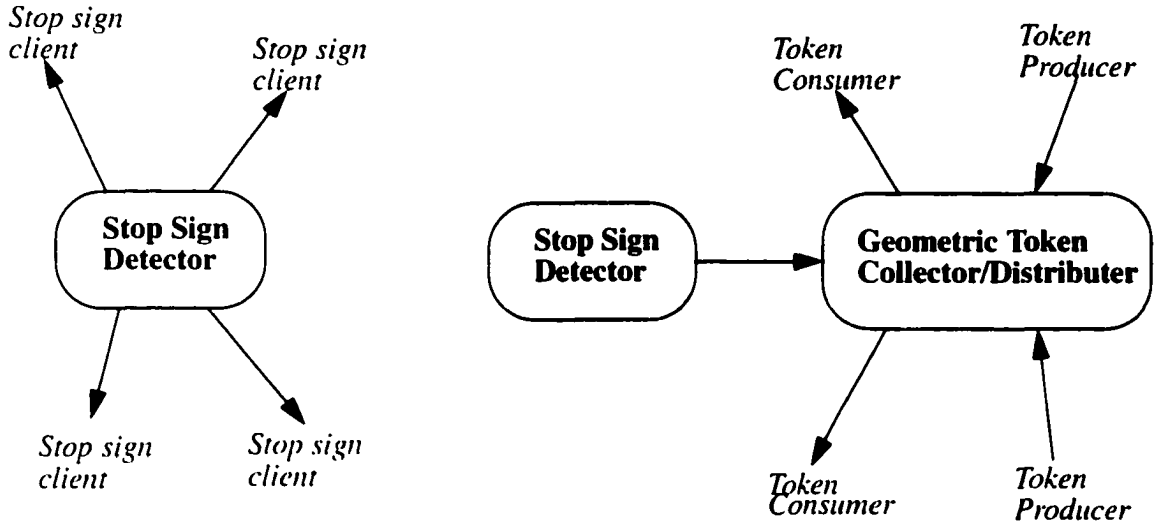


Figure 12 Two architectural paradigms

2.6 Conclusions

Mobile robotics is a research and development field, and software methodologies that engineers use to build payroll systems are not completely applicable. In the classic waterfall the development of a software system is broken into completely separate stages of analysis, design, coding, and testing. In the most conservative form, each of these stages is completely separate, i.e., all analysis must be done before design begins, all design must be done before coding begins, etc.

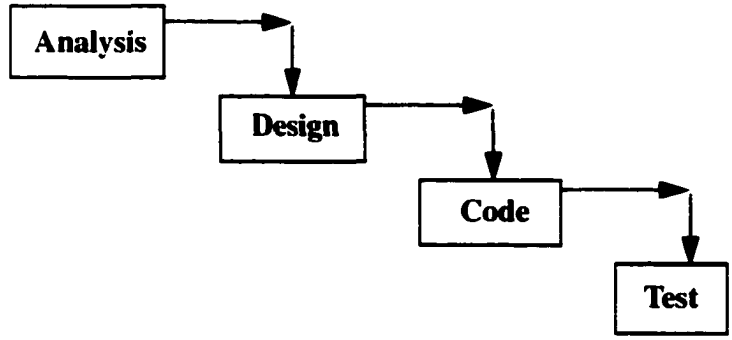


Figure 13 Waterfall development cycle

In reality, the pure waterfall approach to software design is almost never used. In the development of most real software, the various stages overlap to some extent. Modern waterfall approaches suggest an incremental approach, i.e., a series of waterfalls (DeGrace and Stahl) or a set of spirals (Boehm). These approaches acknowledge that the system design and analysis needs to be incremental, and affected by testing with prototypes.

Unfortunately, these methodologies do not address all of the needs of building large research and development systems for mobile robots. There are several features that distinguish building a mobile robot system from building a system in a more stable domain:

- *Volatility of components.*

  In a mobile robot system the major components that actually make the system work, the road followers, the obstacle avoidance systems, the knowledge integrators, are constantly being changed and updated as research on them continues. The structure and working of each "module" of the system are research topics in and of themselves. In a typical software system, the components are relatively stable: once implemented they stay the same barring minor bug fixes.

- *Need for early results.*

  In a mobile robot system there much pressure to produce immediate, tangible results. External pressures can come from hard deadlines in terms of demonstration schedules for funding agents, but they also come from the fact that each component of a mobile robot system is someone's focus of research. Researchers working on road following systems do not want to have to wait six months for a comprehensive system design before starting to work on demonstrable results. There are valid internal pressures as well: real design and analysis cannot be done without information that can only be gained from implementing prototypes, so early results are needed to show whether a system design is feasible or not.

- *Instability of requirements.*

  As the system builders find out what really can and cannot be done, the end requirements of a mobile robot system have to change and adapt to reality. In an ideal software engineering problem, there are stable requirements for tasks, but in a dynamic, research

and development field such as mobile robots, stable requirements are nothing but a dream.

In a mobile robot project, all four of the stages of the waterfall should be going on at the same time. There should be coding starting from the first day of the project as prototypes of components are built or adapted, and system analysis going on through last day, as the designers learn more about the real requirements of the task. There cannot be simple "iterations" of analyze/design/code/test, there must be support for continuous, concurrent evolution of the software architecture with a minimum of work required to reintegrate previously built components into the new architecture.

Rather than stabilizing the systems view of the components as is done in most software reuse approaches, I provide tools which stabilize the components view of the system. A set of reconfigurable interfaces provide an individual system component with a consistent view of the rest of the system, isolating the individual component from the architectural topology and infrastructure. Thus, the system designers are more free to make radical changes in the architecture as new information about how the world really works comes in from the component builders who are free to develop and test in both stand alone situations and in integrated situations.

In essence, the reconfigurable interfaces provide a means for structuring the interaction between module programmers and system integrators. As the system architecture changes, the system integrator will release new libraries and configuration strings to the module programmers as they need to integrate their modules into the new architectures. If there is such a qualitative change in the information needs of the system that new capabilities are required of some of the modules, then it is time to specify or augment the reconfigurable interfaces for those modules. Similarly, if a module's need for information changes radically then once again, the reconfigurable interfaces provide the medium through which to satisfy the new needs. The reconfigurable interfaces provide a buffer between the two to account for much of the continual change that is the very nature of a mobile robot system from top to bottom.

Chapter 3 Architectures and tasks

3.1 Why architectures?

Early mobile robots did not have a real architecture. They typically did one thing at a time, such as first look, then build a map, then plan a path, then move. This meant that they did not have to deal with resource contention or control flow or most of the other architectural issues. Since they usually had a single perception algorithm, there were no conflicts in their world map. And since they were built by a single person, any architectural issues that did arise could be handled intuitively in the designer's head.

Times have changed. Current mobile robots often have many sensors, such as sonar, video, ladar, inertial, etc., and many ways of processing each sensor. They almost always move and sense and plan and coordinate with other vehicles and communicate with remote controllers and aim sensors, all at the same time. And a really big project, such as the Unmanned Ground Vehicle, can be built by hundreds of engineers at dozens of institutions.

Furthermore, there is a growing desire for some kind of standardization. In theory, at least, a component of a mobile robot ought to be usable on more than one robot. Perhaps a motor controller for a walking robot could also control a steering wheel on a wheeled robot, or maybe the map representation used for a swimmer could serve just as well for an unmanned air vehicle. Any such software reuse demands, of course, a set of common specifications.

All of this has led to the development of architectures. But no single dominant architecture has emerged. Quite to the contrary, a heated discussion has been ongoing for the past decade over the form the architecture should take, and architectures vary along many dimensions. One school of thought says that architectures should be hierarchical. A different viewpoint says they should be organized according to independent capabilities. Others propound nested control loops, or fuzzy logic, or central blackboards, or finite state machines, or even matrix organizations.

Which is best? The arguments surrounding that question have been long, heated, and typical of the "religious warfare" that results from clashes of strongly held opinions in the absence of sufficient facts.

Intense ideological arguments over the form of a general robot architecture is at best premature and at worst futile. The robotics community must acknowledge that no one architecture is a perfect fit for all tasks, and that different tasks have different criteria for success which lead to different architectures. Furthermore, there is too little experience with working useful robotic systems to even divide the field into meaningful subdomains, let alone declare architectural standards for subdomains. The field of robotics is developing too rapidly and too dynamically to know if the tasks we are attempting to solve today are the same as the tasks we will be attempting to address in the next decade. If the tasks change radically, the best architectures to address those tasks may change radically.

The emergent architecture approach side steps the problem by attempting to separate the work of module developers from system developers. The reconfigurable interfaces in an emergent architecture allow the module developer's view of the system architecture to remain stable while letting the system developers experiment with software architectures as the needs arise. Thus system developers can create systems that are appropriate for the current tasks rather than being constrained by a single grand unified architectural scheme.

In this chapter I present some general guidelines for how an architecture should emerge from task requirements. The strongly held ideologies of system architecture in the literature should not be ignored, but rather they should be learned from. I present several extreme architectural approaches, how they relate to each other, and how they relate to various classes of tasks. Finally I present my benchmark tasks and what is required of architectures to make them work well.

3.2 Design tenets

3.2.1 Address the needs of the current task

The fundamental axiom of the emergent architectures approach is that there is no single permanent architecture for robot system development. Thus, my first architectural design tenet is to develop and use the architectural infrastructure and resources that address the needs of the task as it stands right now.

I break task requirements into three categories: functional, developmental, and organizational.

The functional criteria involve the factors that come into play when the system is running. The functional criteria can boil down to one question: Does the system perform the task to the desired specification? Questions to be asked when applying functional criteria include,

- What are the "critical" functions?
- Are they operating fast enough, robustly enough, to perform the task?

An architecture must support the debugging and incremental evolution of a task. For an emergent architecture built of components that use the reconfigurable interfaces described in earlier chapters, the issues of developmental criteria are not as acute as they are in systems which use a permanent, fixed architecture which must be the framework within which all development and changes must occur. A system which makes good use of the reconfigurable interfaces can just change the architecture as the real demands of the system are recognized through experiments in the real world. Similarly, the reconfigurable interfaces themselves make powerful debugging tools, making the independent development and testing of components flow more easily into the integrated testing of the components within the real system. Developers can use the reconfigurable interfaces to put components in specialized "debugging" architectures which are stripped of everything except what needs to be debugged.

Organizations that build robots have needs, goals, and structures that have direct implications on the form and content of the robotic systems that they build. Organizational criteria are significant because the better an architecture fits an organization, the more likely that architecture will get built and will succeed at its task. The importance of organizational criteria can be seen in the

uncanny resemblance of the structure of generic mobile robot architectures to the structure of the groups that propose them. A prime example of this is the hierarchical mobile robot architectures proposed by government organizations such as NIST and NASA. The organizations are strictly hierarchical, and the architectures are strictly hierarchical, with power and ideas flowing from the top and getting further and further elaborated through the descending layers of hierarchy. The resemblance is not coincidental, but reflects the realities of how systems get built and developed in hierarchical organizations, i.e., in a top down, hierarchical fashion.

At another extreme there is the close resemblance between the standard "university" architecture, i.e., a centralized integrating/reasoning module with many subordinate modules communicating to and through it, with the structure of the standard "university" research group, with a single "advisor" and many subordinate graduate student and staff. Once again, the resemblance is not coincidental, but reflects the manner in which systems are developed within that organization.

3.2.2 Address only the needs of the current task

My second architectural tenet is to retain **only** the architectural infrastructure and components that address the needs of the current task as that task stands right now.

Often, as the needs of a task change and adapt, features and components that supported the old needs are not longer needed. In a traditional architectural approach, in which a particular architecture is entrenched as the standard of interaction for all modules, the huge investment of time necessary to change the architecture leads to components and features that do nothing, but which are necessary to run the system. These vestigial components and features can introduce unnecessary bugs and overhead, and can complicate the debugging and development of systems that perform the current set of tasks. In an emergent architecture, the investment required to change the infrastructure or run time structure is much smaller than with the traditional approach, so such vestigial components and features can and should be promptly pruned out of the system.

An even more insidious problem is "planning for the future" with the architecture. Developers creating a traditional architecture recognize that changing the architecture is a huge amount of work, so they often take a guess at what the "ultimate" system will need and build these assumptions into their architecture. The problem is that often these guesses are wrong, and not only is the

resulting architecture not optimal for the current tasks, but is also totally inappropriate for the real tasks of the future.

The Navlab group at the Robotics Institute learned this lesson the hard way early in the project. One of the assumptions we made was that the fundamental issue of outdoor mobile robots was geometry. We believed that any future successful integrated outdoor mobile robot would hinge on a standardized method of sharing and reasoning about geometric information. The result was an architecture which had at its center CODGER (COmmunication, Database support, and GEometric Reasoning)[Stentz, 1990]. CODGER provided powerful tools for relating and reasoning about geometric information, and all components of the system used CODGER as a geometric "blackboard" to communicate geometric information discovered in the real world. Unfortunately, in our real systems the geometric reasoning and integration power of CODGER was hardly ever used, and the overhead incurred by the possibility of using these capabilities were crippling. The months of work that were invested in the general capabilities of CODGER that might have been useful someday were scrapped when we were forced to spend months changing over to a new architecture that supported the tasks we were actually attempting at the time.

3.2.3 Architectures are guidelines, not gospel

The emergent architectures approach does not reject architectures, it just acknowledges that they will change from task to task and over time to best fit what a system needs. If anything, the emergent architecture approach makes designing a system much easier than the traditional approach. In most projects, the initial design meeting to produce an architecture is a tension filled stressful event, because everyone involved knows that if a wrong assumption is made, or something is left out, the architecture will be "flawed." With a flawed architecture in a traditional project there are two choices: live with the flaws and the resulting impaired system or invest an overwhelming amount of work in correcting the flaws.

In an emergent architecture, the architecture that comes out of the initial design meeting is known to be a strawman assembled without knowing all of the facts. The initial architecture is a starting point for the system, and is expected to evolve when it meets the real demands of the task. The reconfigurable interfaces of the emergent architecture approach provide a mechanism for

repairing shortcomings and even changing architectural paradigms in the middle of a project with as little impact on the progress of the project as possible.

3.3 General architectures

There are about as many variations of proposed generic architectures and architectural standards as there are mobile robot researchers. I will present three carefully selected extremes in a space of possible approaches: the classical centralized robot architecture, the completely distributed reactive architecture, and the hierarchical levels of abstraction architecture. The reality is that architectures appropriate for most real tasks seldom fall on the extremes, but reside in the continua between the extremes. I will pick two of the relevant dimensions of task requirements, i.e., one being how important and possible world modelling is for the success of the system and the other being how important and possible precision is for the success of the system, and show that these three architectural approaches make up the extrema of the continuum defined by these two dimensions.

3.3.1 Classical architectures

In the early days of robotics it was thought that the way to solve the "mobile robot problem" was by bolting together the new fields of control theory, pattern recognition, and artificial intelligence, with AI being the centerpiece of the system. Unfortunately, a mobile robot is more than just the sum of its parts. The earliest application of this straightforward method such as Shakey the robot and its immediate descendants[Nilsson, 1984] all have a fatal flaw: They all assume that the world is easy to model. Shakey's planning system, STRIPS, is a straightforward implementation of means-ends analysis[Newell and Simon, 1963]. This seminal planning method uses a recursive goal decomposition method to break the mission goals down into a series of primitives that the robot can execute using its control loops and pattern recognition systems. This method assumes that the planner has modeled everything relevant about the environment, that the environment doesn't change during the execution of the plan, and that the robot can execute the plan. None of these assumptions hold true in the real world.

Shakey's philosophical dependents all share the axiom that the central issue in mobile robots is cognitition, i.e., the manipulation of symbols to maintain and act on a world model, the world

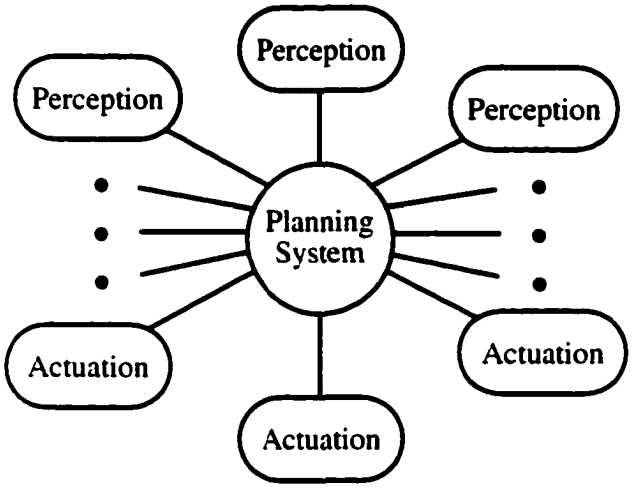


Figure 14 Classical architecture

being the environment with which the robot interacts. The problem of modeling the world is solved by gaining more and more information for the center about the world, but the modern approaches are much more flexible than Shakey. The idea of a plan as something static has been slowly replaced by "blackboard" architectures[Hayes-Roth, 1985], which accrue data about the world and make immediate decisions based on both variable a priori goals and a changing world. Architectures have been proposed and implemented which allow for seamless changing of goals and error recovery[Georgeff and Lansky, 1986]. There are architectures which learn, i.e., which adapt their internal models in a variety of ways such as "chunking"[Laird et al., 1987] or case based reasoning[Carbonell and Veloso, 1988]. All of these architectures still share a common view expressed explicitly in some of the early papers on blackboard architectures: the planning system is at the center. The planning system gathers information to update its world model and issues commands that change its state in the world. The modern systems are more flexible in how they can update their world model, and more flexible in how they can adapt to the reality of an imperfect robot operating in an unpredictable world, but this is really just a quantitative design evolution rather than a qualitative difference in philosophy from the STRIPS approach

All of the classical architectures make the same assumption: that planning, coordination, and task execution is the critical problem in mobile robot tasks. This idea that planning is the critical problem permeates every level of the architecture. Every bit of information in the system is integrated and distributed through the planning system, since every bit of information is needed to

update the world model. The role of any perception modules is reduced to "virtual sensors," which can be added in with the ease of physical sensors such as cameras and sonars.

The classical architecture is a good model for cognition, in which all information is brought to a center, processed, and the result sent out to be acted on, but is not necessarily a good architecture for mobile robots in general. The classical approach works well for tasks in which deliberation and optimality are more important than reaction and reflex. The sacrifice that must be made in a classical system can be summed up in the word deliberative: deliberative means that the system can afford to take its time and think about things before acting. If the system can model the world well enough, and the world obeys those models, and the system can get information to integrate in the central planning core, then a classical architecture will be a good choice for a task.

3.3.2 Reactive architectures

The driving issue in reactive architectures is a negation of the fundamental tenets of the classical approach. Where the classical researchers believe that all problems will be solved if the model that is maintained in the center of the system is just updated and corrected with more and more data, the reactionaries deny the usefulness of a central world model. As Brooks, the most extreme of the reactive camp, says, "the world is its own best model[Brooks, 1986]. Brooks makes extreme claims that systems do not even need to have any internal state or to communicate through symbols at all. Other reactive schemes such as Rosenchein's situated automata[Rosenschein, 1985] or Agre and Chapman's Pengi system[Agre and Chapman, 1987] do moderate this extreme stand somewhat, but all of the reactive systems have a fundamental similarity: The world model is not something explicitly represented and internally maintained and manipulated, it is inherent in the design. Reactive systems are based on reaction to the environment. In their systems, intelligent action "emerges" from a system of behaviors designed to act concurrently on data coming from the real world.

Figure 15 shows the structure of the building blocks of a subsumption architecture, the original example of a pure reactive architecture for mobile robots. Simple modules process input signals that come in over "wires" to produce output signals that either go to other modules or to actuators. A system designer hooks up a network of these modules in a "control layer," which implements one level of competence. For example, the most basic level of competence for an

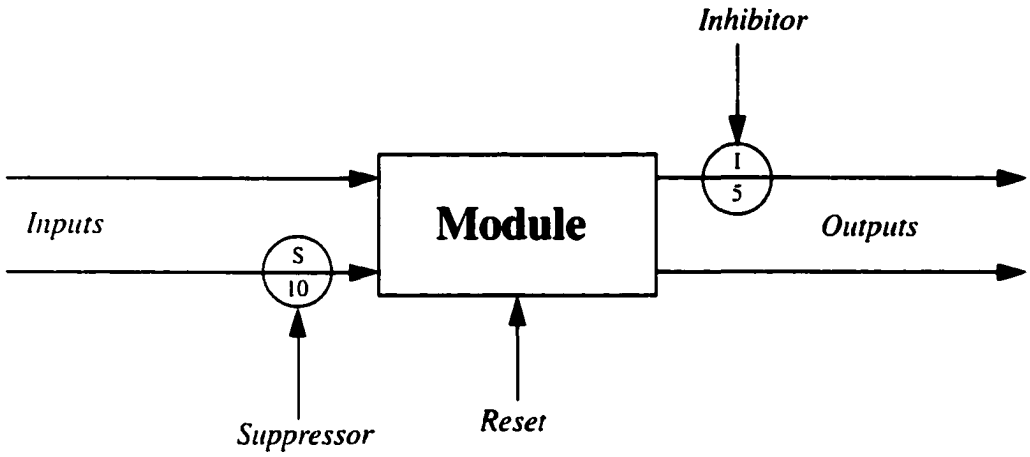


Figure 15 Building block of a subsumption reactive architecture

indoor mobile robot with sonar sensors is to avoid running into obstacles. Further levels of competence interact with the system by suppressing the lower levels inputs and inhibiting their outputs to modify, or subsume, their functionality. The system designer for the indoor mobile robot task can add a "wandering" control layer, which modifies the inputs and outputs of the avoidance layer to wander at random while avoiding obstacles. The wandering task "subsumes" the obstacle avoidance task, i.e., the wandering task is a superset of the obstacle avoidance task and both "tasks" are happening in parallel. The system designer can add more levels of competence as they are developed and tested.

The subsumption architecture works well for tasks that must work in an extremely unpredictable and dynamic world. Since there is no world model, the system does not make assumptions about the world that can prove disastrously wrong. The system is constantly observing the world and reacting to what it perceives, rather than what it thinks is in the world according to any a priori model.

So, for tasks that take place in an extremely hostile and unpredictable world, the extreme reactive approach of subsumption architecture is necessary, but is the world necessarily that hostile? The classical mistake is to assume that a complete model of the world can be made, and complete predictions of the results of the systems actions can be made. Subsumption goes completely in the other direction and says that there can be no prediction of the impact of the system on the world, so all action must be predicated by sensing.

By rejecting internal reasoning about world models, reactive systems lose the ability to ration limited resources. World models might be inaccurate, but they represent a best guess about the world that is supposed to be accurate most of the time. If a world model is mostly accurate, using it can improve the performance of a perception module most of the time, and by the pragmatic principles of system design that I use, if maintaining some internal representation about the world helps the overall system perform its tasks more than it hinders the task, that is the approach that should be used. The success of the immediate task is more important than adhering to any "pure" ideology or architecture.

3.3.3 Hierarchical systems

A hierarchical approach to systems has also been promoted as a general robot architecture. In a hierarchical architecture the system is divided into layers, which represent different levels of abstraction and throughput. At the "top" are the abstract mission goals and plans which are processed on the order of once a day, and at the "bottom" are the control loops which interact with the sensors and actuators on the order of once a millisecond. Each layer issues commands at its own level of abstraction to the lower levels. The lower levels then do their best to elaborate and carry out the commands and return feedback on how well they are doing to the higher levels.

The purest example is NASREM, which was originally designed as an architectural standard for telerobotics[Albus et al., 1987]. NASREM consists of six layers arranged hierarchically, from the bottom layer which actually controls the actuators and sensors to the top layer which deliberates about mission level tasks. At each layer the classical, model based architecture is duplicated to link perception and action, although at the lower level the "world modeling" will not be done with symbols and reasoning but by using the tools of control theory such as feed forward modeling.

The NASREM architecture avoids many of the bottlenecking problems of the monolithic classical approach. Instead of having one general center for reasoning, information is combined in world models at each "level" using the appropriate representation. Ideally each level is in as tight as possible a loop with inputs and outputs to satisfy the commands of the higher layers. As we move up the hierarchy the loops become slower and the data transforms from signals to symbols.

Hierarchical architectures such as NASREM still make the assumption that the best way to interact with the world is through manipulating and reasoning about world models, although they do acknowledge that there must be multiple different world models to reason about different aspects of the world. What a NASREM system can achieve by using predictive models of the world is extremely high precision. Every layer has a model for what will happen in the world given a certain set of inputs and outputs, and it is up to the layers below to make sure that what was expected happens, precisely.

Hierarchical architectures are appropriate for tasks that take place in a predictable environment and require high precision. NASREM itself originated as a standard for telerobotics. The goal of a telerobotics system is to take the commands of a human and precisely and repeatably execute them. Any problems in implementing the human commands can either be viewed as "disturbances" that the system can overcome or as errors that require error recovery and reissuing of commands that meet the new situation, once that situation is modeled and predicted. As well as telerobotics, hierarchical architectures are appropriate for jobs such as robotic painting and welding, in which the criteria for success is the high precision, repeatable, covering of a known surface.

The main failing of NASREM, and other architectures like it, is that the architecture drives the system rather than the system driving the architecture. It is an a priori, artificially imposed taxon-

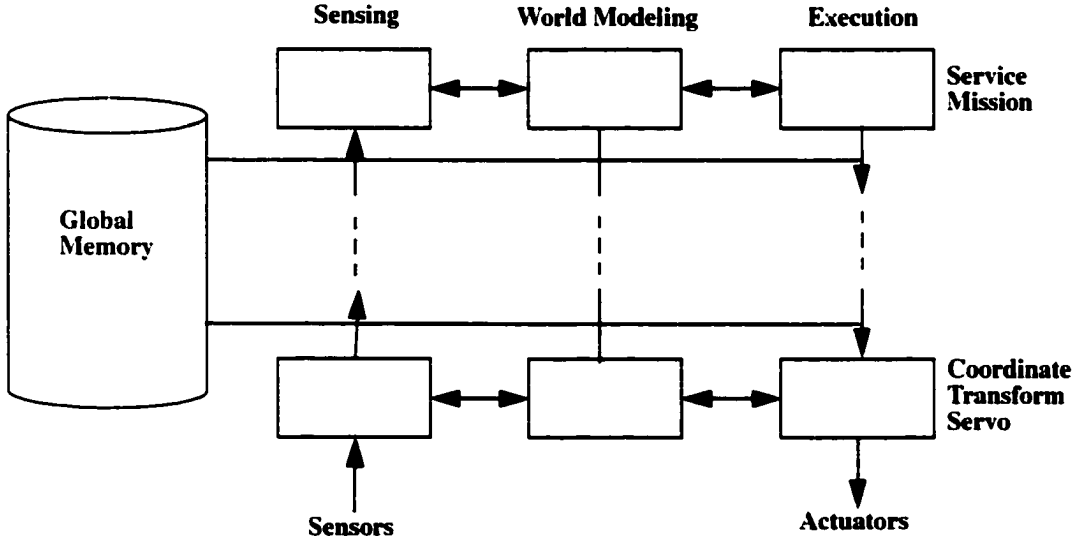


Figure 16 NASREM hierarchical architecture

omy of the system modules that serves to restrain them rather than support them. The way every module in the system is structured is not defined by the task needs, but by where it fits in system architecture. Of course, the NASREM designers state that modules are not simply limited to the sensing information from the layers below and the execution orders form the layers above. There is the "global memory" through which any module can leave messages for any other module. The question then becomes, if every module is really connected to every other module on any layer through the global memory, of what use is the elaborate hierarchical layering system? With full connectivity NASREM becomes simply a tool for describing modules in a system after the system is built rather than an architecture that is useful for developing a system.

3.3.4 Architectural extremes and continua.

There are many dimensions upon which to classify the tasks that a given robot system must address. One important evaluation dimension is the level of world modeling required by and possible for the target tasks, and another is the level of precision required by and possible for the target tasks.

At one end of the world modelling axis is the reactive systems which totally reject world models. At the other end are the classical systems, in which maintaining a world model is considered the central jobs for the success of the task.

For example, Carnegie Mellon's Ambler is a large, legged, statically stable, slow moving robot which is designed to be the prototype for a completely autonomous Martian rover[Bares et al., 1989]. For the Ambler's task a central issue is optimality, in where and how to move and place the legs, in how to ration limited resources, and in how to diagnose and react to problems. At the reactive extreme is Ghengis, which is a small artificial insect designed to explore a completely unstructured environment. Ghengis' task requires robust, continuous action in the face of a dynamic, unstructured, and unknown world.

Most tasks fall in between the two extremes of deliberation and reactivity. The questions that has to be asked is how much deliberation is necessary and how much can there be? In the Ambler system deliberation on the globally optimal course of action to take is vital, and the demands of the task, i.e., a slow, stop-and-go, walker, were such that the bottleneck of centralizing all infor-

mation for deliberation does not hinder the success of the robot. In the Ghengis system, the success of the task comes from quick, continuous reaction to the environment. There is no need and no time for deliberation or reasoning, and a subsumption architecture is perfectly appropriate.

One way to look at hierarchical architectures is as a way to mix the advantages of deliberative and reactive systems. Pure hierarchical architectures such as NASREM strictly separate the world of symbols from the world of signals on different levels. In a hierarchical system, symbols derived from the mission goals and world models get translated into commanded sets of signals, and the patterns from the world are used as error signals to ensure that the desired output is achieved at high precision. Patterns that can be codified as symbols are passed on up to the symbolic layers for incorporation into the world models. The pure hierarchical approach represents one end of another spectrum that ends with the reactive approach: precision in a predictable world vs. robustness in a unpredictable world. In a pure reactive system there are no "commands" coming down from on high, goal-directed behavior that solves a task emerges from the interaction of independent agents. In a reactive system, no one component can predict what another component will do with the signals that are being sent, whereas it is vital that one level in a hierarchy be able to predict what the lower levels will do with their commands. Precision requires precise knowledge of the world, precise predictions of the results of actions, and precise actions in the world.

Most tasks will fall somewhere in the middle of the precision spectrum. The questions for any task are what level of precision is necessary and what level of precision is possible? Painting a car is a task which requires high precision and predictive control in a world that can be fairly well known ahead of time. Any deviation or lack of precision will result in failure, but precision is possible because the world is stable and easy to model. On the other hand, for Ghengis no real precision is necessary or possible: there is no need for precision in wandering around an unknown, dynamic, and unpredictable world, and in fact, the unpredictably of the world would make precision impossible.

3.4 Benchmark tasks

The benchmark tasks are similar in that they both require large outdoor mobile robots, and they both address practical problems that have immediate applications by users. Both tasks can

and should share a fair amount of code, but the needs of each task differs enough that each task will require a different software architecture to succeed.

Since both tasks need to be implemented by the Navlab group, both will operate under the same set of organizational requirements.The organization of the group is a loosely coordinated set of researchers, each with their own goals and tasks. The demands of our organization are that we produce individual Ph.D. thesis projects which are developed, tested, and demonstrated independently, and every now and then are integrated into a whole system. That is the development process that our organizational requirements demand.

As with any organization, our approach to architecture reflects this development process. Rather than have one standard architecture under which all work is done, we have separate architectures for individual components and sub-tasks, which then have to be integrated into another complex architecture when we need to combine the functions of the individual components. The emergent architecture approach with reconfigurable interfaces arose from the real needs of developing systems this way

3.4.1 Infantry scouts

One of the most dangerous jobs in the military is that of a scout. A scout must go in front of the advancing troops and look for enemies. If the scout finds the enemy, then that information can save many lives. Unfortunately, if the enemy finds the scout, the scout is in serious jeopardy. The motivating task of the Unmanned Ground Vehicle program is to put a platoon of semi-autonomous scout robots under the command of a single human supervisor who can direct the coordinated operations of these scouts from a safe position. The goal is to multiply the effectiveness of the trained human army scouts while reducing the risk to them.

To build a completely effective semi-autonomous infantry scout platoon, we must deal with issues of planning, perception and coordinated action for both mobility and reconnaissance. A complete infantry scout has to be able to get to the enemy, detect the enemy, report the enemy, and get away from the enemy while coordinating with the human scout platoon leader and the other members of the scout platoon. At the Robotics Institute, we have focused on the sub-task of the mobility issues involved in a single semi-autonomous infantry scout.

Figure 17 UGV testbed

The single semi-autonomous scout task involves combining and sequencing many different modalities. A fundamental part of the task is combining different information sources to decide where to drive and how fast to drive. These sources of information include road following systems, local obstacle avoidance systems, teleoperation systems, and global route planners. The system has to be flexible enough to incorporate new information sources as they are developed. We also have to be able to sequence the combinations differently at different stages of the mission. For example, when the vehicle is approaching the "front line" on relatively well known roads we will want to drive using just road following at a fairly high speed, but at another we will be combining information from a strategic global route planner and a local obstacle avoidance system.

Our goals for this task are to have a system that can drive twenty to twenty-five miles per hour on well known roads and 3to 10 miles an hour in unknown territory. We build and test our systems on a converted army HMMWV vehicle which has four Sparc 10's running UNIX and a 68040 board running VxWorks for computing, and an array of video and range sensors. We share many of the architectural ideas and individual components that we develop with Lockheed-Martin, the company which is building the system for the full, multi-vehicle reconnaissance task.

For the primary benchmark task the critical algorithm is the integration of the knowledge sources. The integrator must simultaneously and smoothly combine information from many modalities, such as obstacle sensors, road sensors, strategic route planners, and safety sensors. The algorithm that combines this information must be effective, so that the system does the right thing, and it must be efficient, so that the system does the right thing in time. If the combining is not effective, the system will not survive and reach its goals; if the combining takes too long then the system either must run slower than desired or take more risks by running faster than it can truly process information. The integrator is an example of a "bottleneck" whose impact the architecture must minimize as much as possible. The structure of the integrator and how knowledge sources interact with it will be the defining feature of this architecture.

Even though most developmental issues can be covered by the emergent architecture approach itself, the integrated architecture does have some developmental criteria. The task is not simply to integrate a fixed set of modalities, but to be able to integrate any number of knowledge sources as they are developed. So, the integrator must have a simple, consistent interface and be able to work with a wide variety of knowledge sources. Knowledge about the world will come from modules that sense what is immediately in front of the vehicle, such as road followers and obstacle avoiders, from a long range sense of strategy to achieve goals, such as global route planners, and from human intervention, such as teleoperation systems. To complicate matters, all of these varied information sources will be producing their information at different rates as well. In addition, in simulation or passive mode the results of the knowledge integration will not be acted upon, while in an active mode the results will be sent to a controller which will actually move and direct the vehicle.

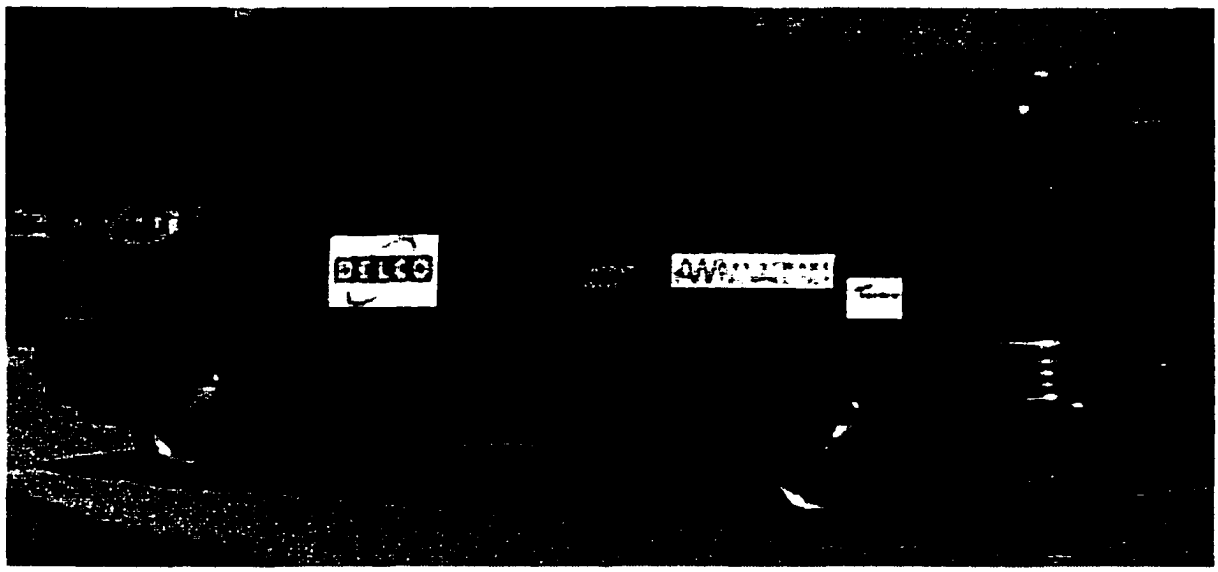


Figure 18 AHS testbed

3.4.2 Automated highways

On of the most dangerous things that a civilian can do is get in the car and drive on the highways. The major enemy on the highways is the boredom, tedium, and inconvenience of a daily drive in an urban environment or a long trip across the country. The Automated Highway System (AHS) program funded by the department of transportation is attempting to address this dangerous task. The dream of the AHS program is to save lives by making driver inattention a non-factor in highway accidents and to reduce travel times and stress by making the daily commute to work quicker and more peaceful. The final version of the AHS system may involve special dedicated automated lanes with hundreds and thousands of robotically controlled vehicles interacting to make driving safer and more efficient. Many issues have to be addressed to make this dream come true: How much of the "smarts" should be in the vehicle and how much in the infrastructure? How will people interact with the system? What will it take to deploy such a system?.

The Robotics Institute's focus in the program has been on single, independent, vehicles that can travel on a highway that has had no special alterations and provisions for autonomous vehicles. The main goal is to drive on highways at speeds up to 65mph, but much of our testing can be done on less structured urban roads at 20 to 35mph. The fundamental problem that we are solving is finding and following roads. For safety reasons we are restricting ourselves now to only controlling the steering with a human either controlling the speed directly or through close monitor-

ing of a standard cruise control. Our testbed is a modified minivan equipped with a single Sparc 10 for computing, video cameras for sensing, and actuators for moving the steering wheel.

For the secondary task, the road follower, the primary functional criteria is obvious: The system must be able to steer on roads at high speeds. Everything in the architecture for this system should be geared towards getting sensor data to the road follower, processing the data, and getting commands from the road follower to the actuators as efficiently as possible. The higher the throughput of the system, the faster and safer the system can drive the vehicle. Anything in the architecture that gets in the way of achieving the goals of the task, i.e., an efficient, robust road follower, must be pruned out.

3.5 Conclusions

The focus in building a mobile robot system should not be to prove an architecture or support an ideology, it should be to perform a task. Thus, the needs of the tasks should drive the structure of the system architecture rather than the choice of tasks being driven and limited to what the ideology or architecture addresses. A good warning sign is if the paradigm for an architecture is constantly acting as a hindrance rather than a help. For example, if in a classical architecture system designers are constantly going around the all important center for "practical" reasons, then there is no reason to hold to the philosophy that all information must be processed by the center. As another example, if in a subsumption architecture designers are producing modules which use symbols and internal state in order to actually get the system built, then the paradigm needs to be changed to something more suitable. Finally, if system designers using the NASREM standard are using the vague "global memory" path to connect arbitrary components together rather than going through the neat official channels, then the pure hierarchical approach is not appropriate for this task. Any architecture provides abstractions and constraints for ordering how a system is put together. Fundamentally, a mismatch between architecture and task can be diagnosed if either the architectural abstractions are not being used or, more importantly, if the system integrators have to constantly work around the constraints provided by the architecture rather than working through them.

Compromise in architectures is not a sign of weakness or chaos, it is a sign of a living, vital system that cannot be put into any one particular box. If a system is built with reconfigurable interfaces, system designers are more free to experiment with different architectures and change them as new capabilities and requirements are discovered. Of course, there are still costs associated with radically changing the architecture of a system, but the reconfigurable interfaces reduce these costs from prohibitive to merely high.

Chapter 4 Architectural Infrastructure

4.1 The building blocks of a system

I call the basic building material of a software architecture the architectural infrastructure. Just as the design of a building is expressed eventually in concrete, glass, and steel, a software architecture is eventually concretely expressed through its architectural infrastructure, in the mechanisms through which information flows and by which components communicate. The materials that make up a building must be appropriate to the its purpose and its environment, or the building fails. The infrastructure of a software system must be appropriate for the task requirements and the software and hardware capabilities, or the system will fail.

With reconfigurable interfaces, I could take the extreme approach of designing an "infrastructure" for each component in the system that is exactly appropriate for that component. Any given component in the system could use abstract data destinations and sources to cover the fact that each component in the system has its own unique method of communicating.

The obvious problem with this extreme is that it leads to a maintenance nightmare. If every component in the system uses a different method of communication, each of those basic information transport mechanisms is a possible source of bugs. Each of those basic transport mechanisms must be learned and maintained by programmers other than the original developer once the original developer has moved on.

Going to the extreme of having a separate infrastructure tailored for each component in the system results in chaos, and the chaos does not even have a tangible benefit. Most components in a system will have similar infrastructural needs, and a single infrastructure can take advantage of these commonalities to serve as the basic medium of information exchange for most of the system. Having a single infrastructure means that there is only one place to go to maintain and look for bugs in the basic transport mechanisms.

A good infrastructure can also provide an additional layer of abstraction for the system software. If the infrastructure has no reference to the underlying transport medium, be it Unix sockets, shared memory, or carrier pigeons, then the implementation of the infrastructure is free to change as the requirements and capabilities of the system change. Hopefully, these changes in the infrastructure can be made without changing the reconfigurable interfaces or the algorithmic code itself.

Of course, the reconfigurable interfaces do allow the system designer to use more than one architectural infrastructure at a time. For example, the overhead involved in making an infrastructure have an identical interface for shared memory transactions and Unix socket messages might be too high to be acceptable for some real time applications, and thus the system designer might build one set of reconfigurable interfaces that uses a shared memory infrastructure and another that uses the system wide infrastructure. Just as the appropriate connections allow a building of glass and steel to have a section built out of brick, the reconfigurable interfaces can be used to smoothly connect two sub-sections of a software architecture that have different architectural infrastructures.

In the remainder of this chapter I present several candidates for the architectural infrastructure of a mobile robot system. The discussion will be a step above the low level protocols, i.e., sockets vs. shared memory, since the implementation of each of these architectural infrastructures can vary according to the needs of the system. I discuss mainly how these infrastructures support the needs of mobile robot systems and how they affect they way in which the systems are developed.

In the end I present the architectural infrastructure I have developed for mobile robot systems, IPT (InterProcess Toolkit) [Gowdy, 1996]. IPT provides simple, consistent point to point communications between the modules of the system. IPT gives modules a simple way to communicate

with each other regardless of CPU type or underlying transport mechanisms. IPT has been tuned to support the module oriented approach to system integration I use in building my robot systems.

4.2 Types of infrastructure

4.2.1 Distributed computing model

One possibility for a mobile robot infrastructure is to take advantage of much of the work done in distributed computing and parallel programming. There is a long and rich history of programming languages for parallel processors and super computers. In these approaches, there are several tasks running concurrently, usually on separate processors, and the tasks have to communicate data and synchronize their operations. Recently, there has been a push for libraries and approaches that will work on many different heterogenous architectures, from super computers to workstations connected by ethernet. These tools, such as the Parallel Virtual Machine (PVM) [Geist et al., 1994] and the Message Passing Interface (MPI) standard [Message Passing Interface Forum, 1994], are candidates for the infrastructure of a mobile robot software system. In both parallel programming and in mobile robot systems we have the need to join together multiple threads of control to achieve a single purpose.

4.2.2 Information oriented infrastructure

There have been many architectural infrastructures developed specifically for mobile robots. Most of these infrastructures have come from workers whose primary interest has been planning and artificial intelligence. Thus, the architectural infrastructure that comes from these workers has the planning or plan execution system as the center of the very infrastructure. All communication in the system is mediated by a central hub, which is also, not coincidentally, intimately involved in the planning and task control.

The underlying theme of these kinds of architectures is that all communication is information or knowledge oriented. Modules produce informational "tokens" and publish them. Other modules watch for tokens of a certain type to be produced, and consume them as they are published. A common metaphor for this kind of communication is a blackboard[Hayes-Roth, 1985]. Components write their output on the "blackboard", and other modules read the writing that appears on

the blackboard to complete the transaction. The blackboard is usually more than just a passive place to post and read information, but is the center of reasoning and planning in the system.

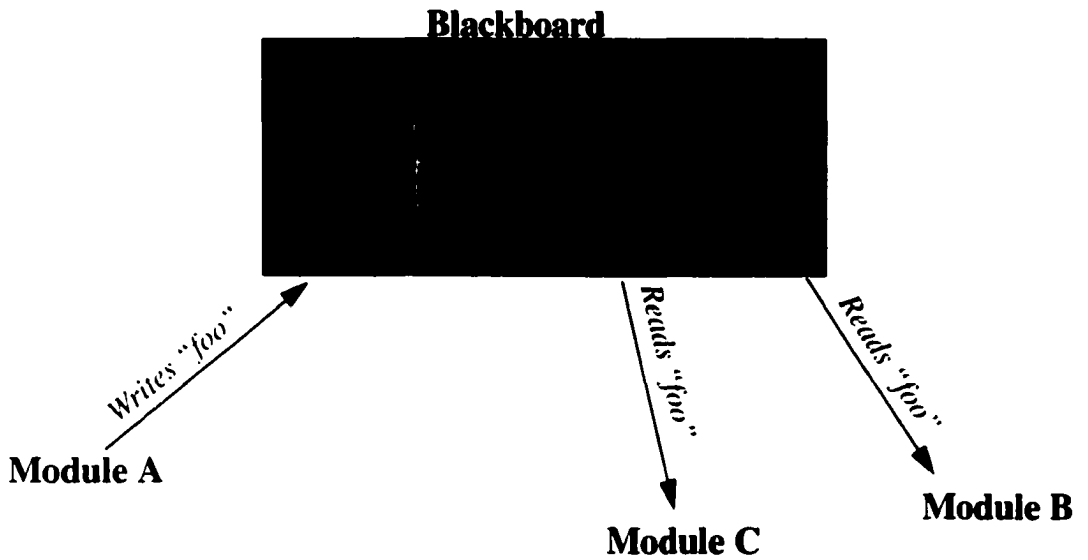


Figure 19 Blackboard communication

Some recent robotic architectural approaches such as TCA (Task Control Architecture) explicitly disavow connection with the blackboard approach, but they essentially provide the same infrastructure for the same reasons [Simmons, 1994]. TCA has a central server which is the means by which all communications is done, and not coincidentally, is the means by which plans are executed and tasks are controlled. Instead of writing tokens into a blackboard, modules "broadcast" messages of a certain type by sending them to the central server. Other modules then sign up to receive messages of certain types by requesting them from the server. The basic quantum of integration is the message type, which is completely analogous to the tokens types of the blackboard systems.

4.2.3 Module oriented infrastructure

In a message oriented infrastructure, the basic unit of integration is the message type or message name. In a module oriented infrastructure, the basic quantum of integration is a module identifier or name. A module, which may or may not be an independent process depending on the implementation, has both an identifier and a well known abstraction, or interface definition. Mod-

ules in the system request connections directly to other modules in the system by name or identifier and interact according to the known abstraction.)

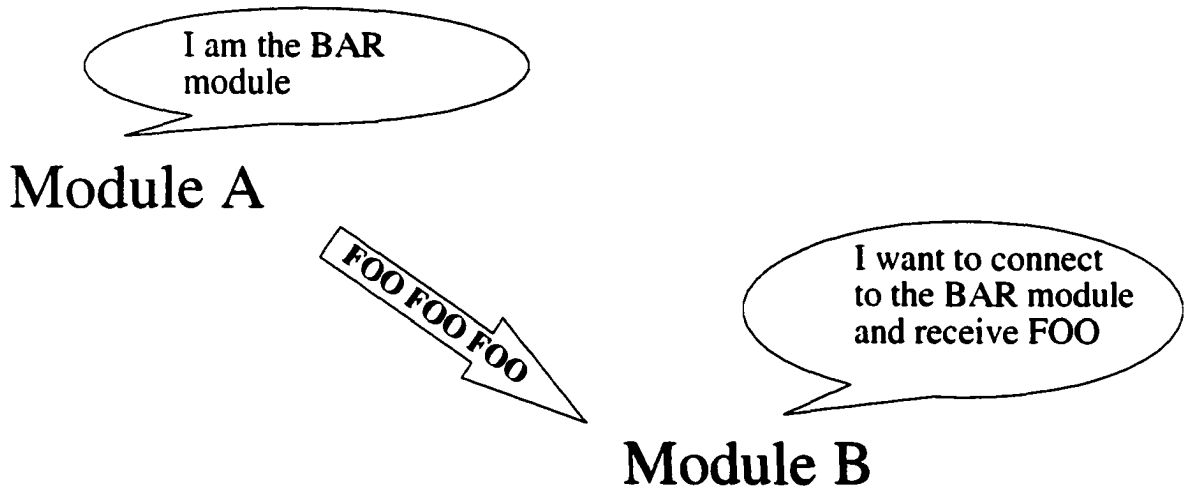


Figure 20 Module oriented communication

Module oriented infrastructures are a superset of object oriented infrastructures, i.e., every object oriented infrastructure can be considered a module oriented infrastructure, but not all module oriented infrastructures are object oriented infrastructures. An object oriented infrastructure is, logically, based on objects. Chapter 2 presents the defining aspects of an object as abstraction, encapsulation, and class hierarchy. Both modules and objects share the idea of abstraction. A module's abstraction is represented by how it reacts to incoming messages from other modules, i.e., whether these messages will be answered as queries, will set off monitoring, or have side effects in the "real" world. Both modules and objects use their abstractions to encapsulate their implementations and hide them from clients and peers. Where modules and objects differ is that modules have no formalized means of implementing a class hierarchy.

Figure 21 shows an example of a simple class hierarchy that may be used in an object oriented infrastructure. A basic "class" is a resource manager. The abstraction of a resource manager is simple: it responds to a lock message by either locking a resource or by reporting that the resource is already locked by another client. Subclasses of resource manager each control a physical device, such as a pan/tilt head, a digitizer, or a steering wheel. A pan/tilt controller is-a resource manager, as is a digitizer or a steering controller. Each of these subclasses inherits the lock method, i.e., the way that the resource task handles a lock message. Thus, an external user of such

diverse objects as a steering controller and a digitizer can use the same code to lock these resources because the objects both descend in an "is-a" hierarchy from the resource manager class. An object oriented infrastructure provides formalized mechanisms for creating and working with a class hierarchy.

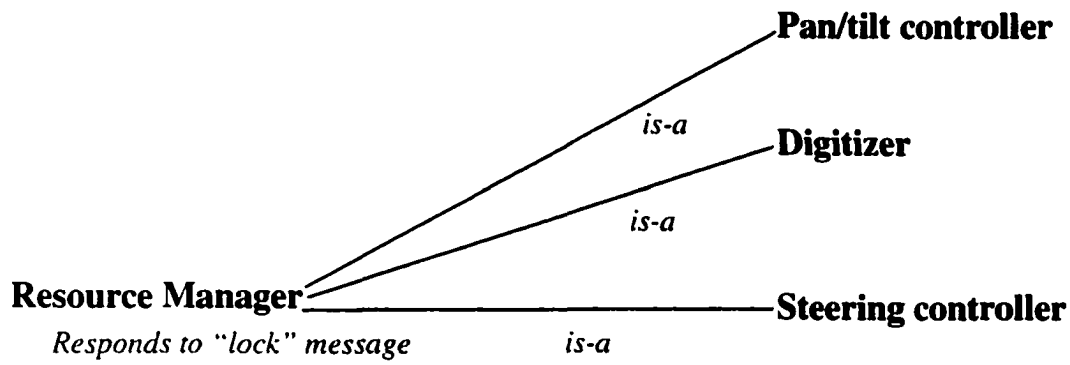


Figure 21 Example class hierarchy

A module oriented infrastructure can take advantage of a class hierarchy, but it does not have the built in mechanisms that a true object oriented infrastructure has. For example, the response to a lock message could be implemented in a library. If a module includes the library and initializes the code, that is roughly equivalent to declaring that an object is a subclass of the resource manager and inheriting the methods for dealing with lock messages. Of course this "inheritance" mechanism does not deal in a standardized manner with such issues as clashing multiple inheritance. Similarly, the mechanisms by which an object does not simply inherit from a superclass but changes the implementations can also be implemented in a module oriented infrastructure, but the means will always be less elegant, more ad hoc, and less standardized than in an object oriented infrastructure.

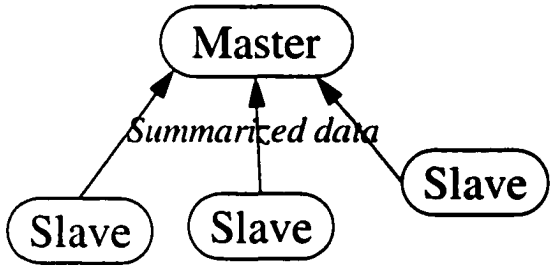
4.3 Choosing an infrastructure

4.3.1 Why not the parallel processing model?

The fundamental difficulty with using interprocess communications tools developed for parallel programming is that the tools were developed for different purposes and with different models

in mind than what mobile robot system developers need. Libraries such as PVM or the MPI standard are designed to make the semantics of programming parallel processes and tasks the same whether the processes are implemented on parallel processing machines such as a Paragon, on a supercomputer such as a Cray, or on a network of Sun workstations connected by Ethernet. Mobile robot software systems are not developed as a single parallel program. Individual researchers build their components in relative isolation, and then the pieces are hooked together. This is in contrast to the normal development of a parallel program, in which a single, sequential algorithm is broken into parallel components.

I find that mobile robot systems require more sophisticated message and event handling than packages such as PVM or MPI provide. For example, PVM is optimized to efficiently implement common parallel programming scenarios such as a master collecting data from many slaves or a single program multiple data (SPMD) system in which many processes process different data in the same way. In both of these common scenarios there only has to be one "type" of message that a process blocks until it receives. This kind of message type simplicity is typical of many parallel programming algorithms, so the simple "receive message" capabilities of PVM are more than adequate. Mobile robot systems usually have a much more complex set of message types, with servers processing many different types of messages and reacting in different ways to each one. In addition, modules in a mobile robot system need to react to more than just messages: other types of events need to be handled such as connections being established and broken or timers being triggered. A simple "block until message received" approach will not be the best way to provide for the needs of a typical mobile robot system.




```
master() {
    initialize_slaves();

    while (1) {
        receive_slave_message();
        process_slave_message();
    }
}
```

Figure 22 Message passing in a parallel programming paradigm

PVM and MPI also assume that there is enough bandwidth in the communication between processes to make the system "look" like a single parallel processing machine. The parallel processing communications toolkits are designed to make a very fast TCP/IP connection between two UNIX machines look like a blindingly fast connection between two nodes of a specialized parallel processor. A real multi-vehicle mobile robot system will often include very low bandwidth links, such as a wireless radio link between two vehicles. The very low bandwidth links will behave radically different than the expected high bandwidth links.

A parallel processing model is not appropriate as the basis for an entire mobile robot infrastructure, but a parallel processing model might be useful for parts of the system. Interprocess communications tools such as PVM or MPI can find niches within a mobile robot system in subsystems for which a conventional parallel processing approach is appropriate, but by themselves they are not good candidates as the basic architectural infrastructure of a mobile robot system.

4.3.2 Message oriented vs. module oriented

A module oriented infrastructure is very close to how communications between components is actually implemented in our current systems, i.e., direct network connections between processes. Any message oriented infrastructure is an abstraction which hides this fact, i.e., that when a piece of information is "broadcast," what is actually happening is a piece of data is being communicated across a TCP/IP connection from one specific process to another specific process. Most candidates for architectural infrastructures present a set of such abstractions which on the one hand may provide extremely useful and convenient functionality, but which on the other hand take the user further away from the actual implementation incurring some level of overhead. The use of an infrastructural abstraction needs to be balanced against its cost, and in addition there is the danger that by hiding the low level functionality behind an abstraction, we can reduce the options available to a system integrator to react to situations for which the abstraction was not created.

An analogy can be made to the debate in the field of computer architectures over Reduced Instruction Set Computers (RISC) vs. Complex Instruction Set Computers (CISC). A RISC architecture gives developers a small atomic set of primitive instructions, while a CISC architecture makes assumptions about what the user needs and provides developers with larger, more complex

instructions that satisfy those needs efficiently. An argument against the CISC approach in mobile robot software infrastructures, i.e., information oriented infrastructures, is that we do not know enough to make assumptions confidently about the needs of users and tasks. We need to have infrastructures that can adapt and be useful as the real demands of tasks and capabilities come to light.

4.3.2.1 Broadcasting and wire-tapping

The key abstraction of a message oriented infrastructure is that a single module in the system only knows that it produces and consumes messages of certain types. That module does not have to know where the data is going or where the data is coming from. Conceptually, broadcasting messages system wide is how all data gets transferred. Since all messages are broadcast, any module in the system can set up a "wire-tap" on communications between any other modules. The designers of the message oriented systems tout this flexibility as a great boon to debugging and development.

This abstraction comes at a cost. Obvious implementations of a message oriented infrastructure would involve actually broadcasting messages system wide, or sending messages through a central repository that servers as a intermediary. Either of these naive implementations will have an unacceptable negative performance impact on any reasonably large robot system I need to build, either by flooding the network with unnecessary messages or by increasing the time to get a message from one module to another by a factor of two. Even more sophisticated implementations of message oriented infrastructure require significant overhead to make what is fundamentally a point to point communications medium "look" like a medium in which information is broadcast anonymously throughout the system.

I argue that for the types of systems I build, not only do the implementation costs of the "broadcast" abstraction outweigh the benefits in flexibility and debugging, but the abstraction goes too far in achieving those benefits. There is no real equivalent of a system wide broadcast in the module oriented paradigm, since every module must know explicitly what modules (or objects) it is talking to, but the equivalent flexibility can be attained given a proper, consistent, object oriented design. In the module oriented paradigm, system wide broadcasts of message types are not only unnecessary, they are the sign of a bad design.

For example, take a boom control subsystem for an automated dirt loading system. If the designer of the boom control subsystem did not define methods for external modules to get at the status of the boom, then that designer did not do a good job. A good object oriented infrastructure will give some mechanisms that ease the "publishing" of this type of information to whatever other modules need it, but the publishing is still done through explicit, point to point connections. In more detail, the boom control module will publish a message "BoomStatus" whenever the boom status changes. A perception module can subscribe to "BoomController" for the message "BoomStatus". When the boom dumps its load, the BoomController sends the status to all modules that request it, one of them being the perception module. The perception module then takes its range image.

In solving the same problem in an information oriented system, the temptation is to "tap into" all dump commands. Thus the perception module would listen for dump commands, and would then would make the possibly ill advised assumption that a "dump" command means a "dump" action. In fact, only the designer of the boom control subsystem knows when a dump is really a dump. The reception of a dump command does not necessarily imply a successful dump.

This example illustrates a pitfall of the message oriented approach: It leads to the idea that all messages are public domain. To use a programming metaphor, this is equivalent to making your code depend on the internal global variables and structure of a library. The internal communication of a sub-system should stay just that, internal, or the reliable, predictable evolution and development of the system is risked. Even apart from this "data hiding" problem, there is the tendency in information oriented systems to take advantage of "side effects." In the boom controller example, developers have to assume that a "dump" command equals a "dump." This kind of undependable side effect is easy to come to depend on in a message oriented system. If a client wants to find out about dumps, it should contact the boom controller, not just listen in on the conversations with the boom controller and guess.

Another question with system wide broadcasting is whether it scales. For example, say there are two pan-tilt heads. The obvious way to control this is with two pan-tilt controller modules. How do the messages being sent to pan-tilt-A get differentiated from the messages being sent to pan-tilt-B? Do we have to define new message types such as "pan-tilt-a-move" and "pan-tilt-b-

move." Is there some mysterious routing going on unbeknownst to the sender and the receiver? In a module oriented infrastructure, the solution is simple, there are two pan tilt controller modules with the same interface and clients in the system must know which controller module they want to talk to.

4.3.2.2 Built-in task and resource control

The abstractions of most message oriented infrastructures are tuned to make task and resource control easier, since the designers of these infrastructures often make the assumption that task and resource control is the key issue to be addressed in building mobile robot systems. These infrastructures provide built-in support for a particular approach to task and resource control, and the assumptions made to ease the job of task and resource control pervade the basic make up of the infrastructure.

Let us take the example of a "resource," such as a pan/tilt head controller. A good information oriented infrastructure will have built in, standardized ways of locking or reserving the use of resources such as the pan/tilt controller. Thus the job of the task planning and execution system is made easier, because it has a standard way of reserving the use of this pan/tilt resource for a single client, thus preventing multiple, conflicting, commands being sent to the pan/tilt head.

The module oriented approach makes no such assumptions about what are the most important issues in the system. This has the disadvantage that the developer has to build the methods and mechanisms for doing task and resource control, but has the immense advantage that the developer can choose the most appropriate methods and mechanisms that fit with the goals and capabilities of the system.

For example, one module oriented approach to resource management is to let resources manage themselves. Say there is a "pan tilt" resource. It is trivial to write code to "lock" in only one user of the "pan tilt" resource. If a developer finds that this code is needed over and over again, then the developer can create a library which adds methods to the current object to make it a "resource." This is one simple interpretation of what the relationship "*pan-tilt* **is-a** *resource*" means: the object *pan-tilt* includes the library which adds the resource management methods. The developer of the "*pan-tilt*" object knows exactly how clients should interact with it, and only

allows that kind of interaction. This means that any task control system will have to know the interface definition of a *resource* object before the control system can control resources.

A standard resource and task control mechanism would be a useful abstraction to have in my mobile robot systems. Not having one incurs obvious risks and costs, i.e., the cost of duplicated effort as several system integrators build separate resource management systems and the risk of multiplying bugs arising from the interactions of these separate resource management systems. The assumption that drives system designers to message oriented systems is that their standard abstraction for task and resource control should fundamentally affect the very manner in which information is exchanged. I do not want to push disputable abstractions into the very infrastructure of my architecture, so I choose to build such abstractions on top of an explicitly module oriented infrastructure.

4.3.3 Why not a COTS object oriented infrastructure?

Using a module oriented infrastructure fits the demands of the tasks I have to perform. The question is, where does this module oriented infrastructure come from? One option is to use an implementation of the Common Object Resource Broker Architecture [Object Management Group, 1993], which is a true object oriented infrastructure developed as a generic software engineering tool. CORBA has the advantages of being extremely flexible, developed specifically for object oriented system development, and available as a commercial off the shelf (COTS) product.

CORBA is a mechanism by which clients can create, manipulate, and destroy objects without regard to how the object is implemented. Figure 23 shows a client making a request of an object implementation. The request is brokered by the Object Resource Broker (ORB). The ORB locates the relevant object implementation and passes on the request, creating or destroying the object implementation as necessary. CORBA formalizes the interaction between clients and object implementation through the use of an interface description language (IDL). The IDL defines what

requests a client can make of an object implementation and how the object implementation returns results to the client.

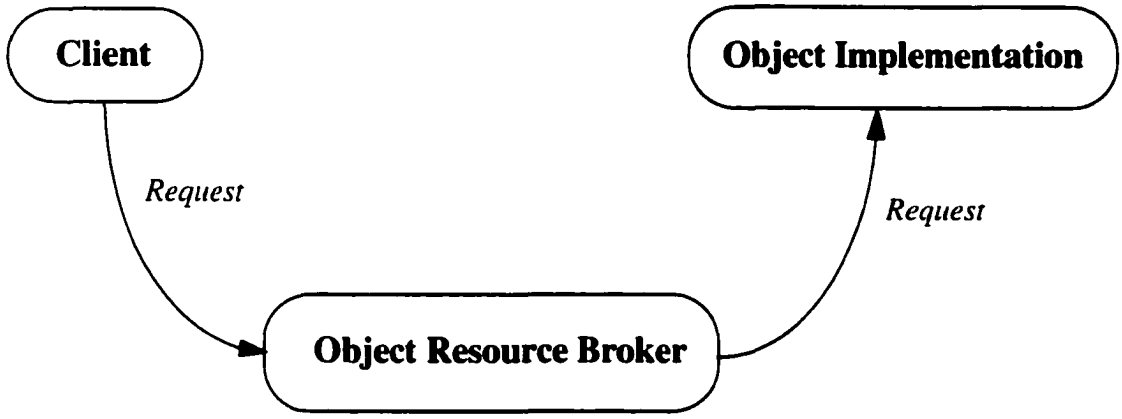


Figure 23 Common Object Resource Broker Architecture (CORBA)

A straightforward implementation of the CORBA standard, with the ORB being a completely separate process, would obviously not be the right choice for a mobile robot system. A mobile robot system might not be able to afford the extra latency involved in sending requests through an intermediate process. Intriguingly, the CORBA standard does not require this. ORB's can be implemented in a distributed fashion, with parts of the ORB resident in the clients and object implementations. Thus, there might be part of the ORB implemented as a process that acts as a "traffic cop," directing the parts of the ORB resident in the clients and object implementations to set up direct connections between processes.

An object oriented infrastructure is a level up in abstraction from a module oriented infrastructure. The abstraction of modules as objects provides mechanisms for ordering and predicting how a module will interact with other modules. The abstraction of object "brokers" frees a system designer from needing to specify exactly what module needs to talk together, instead requesting classes of objects which the broker then provides. Unfortunately, CORBA, and other COTS object oriented infrastructures, are too big and too new. At this point, CORBA is still being developed and is still maturing. It is also maturing as a general solution to distributed software development. What I need is an infrastructure that is smaller, less complex, and more focused on the problems found in developing mobile robot systems. I can more easily port a small infrastructure to the various platforms and communications protocols needed to make the systems work than I

can port a COTS set of tools and applications. The small, simple infrastructure can have exactly the level of abstraction that is needed, and nothing more, rather than picking the features that are needed of CORBA and ignoring, while still needing to port, all of the other features.

Object oriented infrastructures such as CORBA can be considered a layer of abstraction over module oriented infrastructures, i.e., they add useful abstractions without necessarily conflicting with the fundamental transport mechanisms. My current systems, which are experimental prototypes with a fixed topology during a system run, do not currently require these abstractions, but as the research matures, such abstractions will be useful. Once the object oriented infrastructures have had more time to mature, they will be very good candidates for mobile robot infrastructures, but for solving the problems I have right now, I only need something simpler and smaller.

4.3.4 Agent oriented infrastructures

Another layer of abstraction beyond the module oriented approach is to consider each module not as an process exchanging messages with other processes, but rather as an autonomous agent exchanging knowledge with other agents. An agent oriented infrastructure would use toolkits developed under the auspices of distributed artificial intelligence, such as the Knowledge Query and Manipulation Language[Finin et al., 1994], as the basis of communication.

KQML provides mechanisms for agents to describe what they know, advertise that knowledge to other agents, and acquire advertised knowledge from other agents. Essentially each agent has a knowledge base which contains facts that the agent knows and assertions that the agent believes, and KQML provides a way for two agents to share this knowledge. The actual interaction between two agents still takes the form of "messages" that goes between two processes. Some of these messages contain data from the knowledge bases, but many of them contain operations to be performed on the knowledge bases as well, such as querying or subscribing to values. So KQML messages have a rich semantic wrapper which not only describes the data in the message, but the action that should be taken on the receivers database (this action is also known as a "performative"). The core of KQML is to represent, manipulate, and distribute the knowledge that an agent has, not the messages that fly back and forth between processes.

KQML could be used to implement a pure point-to-point architectural infrastructure, as seen in Figure 24a, in which agent *A* knows about agent *B*, and asks it whether *X* is true or not, with agent *B* replying. An advantage of KQML is that it is easy, and in fact critical, to set up "facilitator" agents which can act as intermediaries between agents searching for knowledge in the system. Figure 24b again shows a simple example in which module *A* and *B*, and *A* in this case needs to know if *X* is true or not, but in this case it does not know that agent *B* knows *X*, so it uses a facilitator agent, *F*. *A* uses a *subscribe* performative to request that facilitator *F* monitor for the truth of *X*. Once *B* tells *F* that it believes *X* to be true, *F* will then tell *A*.

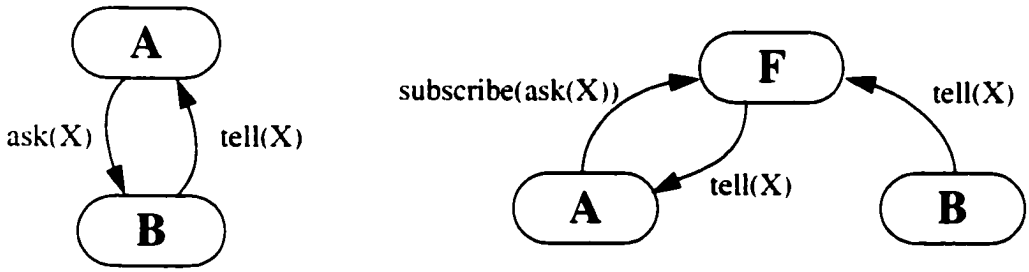


Figure 24 KQML interactions

Agent oriented architectural infrastructures are at an even higher level of abstraction than object oriented infrastructures, i.e., rather than abstracting the modules as objects passing messages it abstracts the modules as agents with publicly accessible knowledge bases. In fact one implementation of KQML uses CORBA as its basic transport mechanism, indicating where each stands on the scale of abstraction. The agent oriented approach comes with its own set of assumptions and overhead costs, and while the agent oriented abstraction may prove extremely useful, it is not necessary for the problems I have at hand. Once the problems I work on mature, an agent oriented toolkit such as KQML will be another architectural infrastructure which will deserve serious scrutiny.

4.4 InterProcess Toolkit (IPT)

I have developed the InterProcess Toolkit (IPT) as a bare bones architectural infrastructure, i.e., a consistent medium for architectural development, for module oriented system development. The purpose of IPT is to provide the simplest possible infrastructure for module oriented develop-

ment that transparently takes advantage of the most efficient transport mechanisms. IPT lets modules connect directly to other modules without regard to how that connection is made, whether it is a TCP/IP connection, a shared memory connection, or even a connection mediated by another module. IPT also hides hardware specific details such as byte order so that modules running on different kinds of processors can interact.

IPT uses many classes of objects to encapsulate the communications functionality. These objects include,

- *Communicator objects*

  Each module that uses IPT creates a communicator object which encapsulates the code for creating connections to other modules and registering and handling messages.

- *Connection objects*

  The connection objects encapsulate the methods for sending messages back and forth between modules. The actual implementation of the object may change, i.e., one connection object may use TCP/IP socket based communication to send data while another may use shared memory, but the user's view of the connection object remains the same.

- *Message objects*

  Modules communicate by using connection objects to send and receive messages. At its most basic an IPT message object consists of a named type (registered using a communicator object) paired with an array of bytes. IPT provides mechanisms for packing and unpacking the raw data from standard structures using format specifiers. The packing and unpacking process allows IPT messages to be sent and interpreted correctly on heterogenous platforms, i.e., the packing and unpacking process automatically corrects for different byte ordering of integers and floating point numbers.

Figure 25 shows gives a simple example of how to use the IPT toolkit to create a connection and send a message across that connection using C++. The first step is to create the communicator. In this example, the module will be named "foo", since that is the name passed to the communicator creation function. In current implementations of IPT, the communicator creation function looks for a "server" process. This server process provides two primary functions,

1. To act as a telephone operator connecting modules together. A module can thus request a

connection to another module by name without having to know what machine that other module is on or how to connect to that other module.

2. To establish a consistent mapping between message type names (which are strings) and message type ID's (which are integers). This mapping means that each message will have associated with it a four byte integer rather than an arbitrarily long type name. Having the server make this mapping ensures consistency across all modules that the server works with

No information flows through the server, so it is in no way a bottleneck, although it is a central repository of module and message names. Modules only interact with the server to register messages and find other modules in the system.

The next step in the example is to register the message "TestMsg." This registration process means that this module can send and receive messages of the named type "TestMsg" and the server assigns an integer to be used when actually sending the message to other modules. In most more realistic examples, the type "TestMsg" will also be associated with a format specification string that specifies what kind of data will be packed into the data of a message of type "TestMsg," but in this simple example there will be no formatted data.

Then, the module uses the communicator to connect to another module named "bar." The communicator object queries the IPT server for the best way to connect to "bar," and when it gets

```
/* create a communicator */
IPCommunicator* comm = IPCommunicator::Instance("foo");

/* register a message */
comm->RegisterMessage("TestMsg");

/* make a connection to module "bar" */
IPConnection* conn = comm->Connect("bar");

/* create a message with a user defined function */
IPMessage* msg = user_defined_message_creation();

/* send a message */
comm->SendMessage(conn, msg);
```

Figure 25 Simple use of the IPT toolkit

this information, creates the appropriate connection object. Finally, a user defined function is used to create a message, and that message is sent to the module "bar." The implementation of the connection object determines how this message actually gets sent, but whatever the implementation, the module programmer still uses the same syntax.

IPT provides more than just a way to send and receive messages. Specifically, it provides support that a module oriented infrastructure needs. First, it lets a module use message handlers. If the module programmer uses the communicator object to add a message handler to a message type, whenever a message of that type comes in, the communicator invokes that message handler. Message handling provides a level of flexibility that is vital to a module oriented infrastructure. For example, message handlers allow a module to be structured as an event loop, with the events that can be responded to specified by what handlers have been registered rather than by a single, fixed set of conditional statements.

Message handlers support managing a module's input flexibly and simply. IPT also provides built in support for managing a module's output through routines for publishing and subscribing to data. A module uses the communicator to declare that it publishes messages of a certain type. External modules can use their communicators to subscribe to data of that type at the publishing module. When the publisher publishes the data type, that data goes to all subscribers. The publisher/subscriber support provides much of the flexibility of the "broadcasting" features of the information oriented infrastructures while remaining within the framework of a module oriented approach.

IPT also recognizes that mobile robot systems are dynamic, with modules going up and down with alarming regularity. IPT provides mechanisms for detecting and handling modules connecting and disconnecting. IPT also has built in support for robust relationships between publishers and subscribers and between clients and servers in the face of modules crashing and restarting.

IPT is implemented in C++ in order to take advantage of C++'s object oriented features to give the "objects" that the end user sees the most flexibility with the most efficiency. Even though IPT is implemented in an object oriented style with an object oriented language, IPT does not force its users into the total adoption of an object oriented approach. IPT provides cover functions

for C so that programmers using C do not have to convert long standing code and approaches to coding.

4.5 Conclusion

The guidelines of emergent architectures applied to an architectural infrastructure would be that the infrastructure should be as complex as the current set of tasks require, and the infrastructure should be only as complex as the current set of tasks require. IPT is an architectural infrastructure that satisfies these guidelines for my tasks.

IPT was not developed for some other purpose, such as demonstrating planning techniques or building better parallel programs. It was developed specifically for the brands of outdoor mobile robot tasks that the Navlab group addresses. These tasks require fast, direct connections between modules using the most efficient possible transport mechanisms. These tasks do not require anonymous communications or built in support for plan execution and monitoring. Thus, IPT is a module oriented infrastructure, i.e., modules connect directly to other modules by name and send messages across that direct connection rather than broadcasting messages to anonymous consumers.

IPT does contain many features that our experimentation has shown our tasks to require. These include:

- Connection objects which encapsulate the implementation of connections between modules so that transport mechanisms ranging from TCP/IP to raw shared memory can be viewed the same way by the end user
- Message objects which can tailor how messages are delivered and processed to maximize efficiency and to hide incompatibilities between modules running on different hardware platforms such as the order of bytes in integers and floating point numbers.
- Sophisticated message handling for message coming in, and support for a module to act as a publisher sending messages to a set of clients.
- Handling of other events, such as connections being broken or established.

IPT is aimed at our current tasks, which, once run, have a fairly stable and constant topology. A system designer setting up an architecture for a task knows what modules are in the system and exactly what modules need to connect to other modules. Having to exert this level of control over the topology is not a flaw for our current systems, but an advantage, since it means the system designer must have intimate knowledge of how information is flowing in the system, which is critical for debugging the system. Eventually the kinds of tasks we will be addressing will require more fluid and dynamic control over the system topology, i.e., how to connect modules together, and abstractions such as object brokers or knowledge facilitators will be necessary, and will need to be explored.

An infrastructure has a significant amount of inertia, so an infrastructure should be appropriate to an entire phase of a project rather than just any single task. Normally a commitment to an infrastructure would be almost impossible to undo, since a change in the infrastructure would mean intimately changing every component in the system. For example, the infrastructure choices I have made that have been driven by the fact we are developing static, prototype robot systems would be incredibly short-sighted if I expected to have to use the same infrastructure 5 years from now. Fortunately, reconfigurable interfaces make the transition possible, though still painful. When the infrastructure changes, most reconfigurable interfaces will have to change, although the modules themselves should remain untouched, so I can afford to make the commitment to a module oriented infrastructure without agonizing over whether I should have chosen an object or agent oriented infrastructure. When our project enters a different phase with different goals, we can reevaluate the infrastructure and move to one that is more appropriate for that phase.

Chapter 5 Real Architectures and Results

5.1 Emerging architectures and system development

The emergent architecture technique embraces the need for change in an architecture. The approach provides tools, i.e., a reconfigurable interface system, for moving major components between architectures as the architectures change and evolve. Reconfigurable interfaces ease the inevitable transitions to new architectural paradigms and allow system components to move from one project to another with the least possible impact while allowing the different projects to explore different architectural directions.

In this chapter I present two real architectures which have emerged from the requirements of two different goals, one an integrated multi-modal cross country navigation system and the other a highway road follower. I show that the reconfigurable interfaces let a system integrator move a vital and complex component between the two resulting architectures simply and with no recoding. The ease with which the component, a road follower, can move between the two very different architectures validates the claim that I can isolate modules from the system.

Furthermore, this chapter demonstrates that the tools used to build an emergent architecture ease the development and integration of components into a larger system. I show how the reconfigurable interfaces reduce the bandwidth of the interaction between system developers and the

component developers, and smooth the path for integration, development, and maintenance of complex components into an architecture which itself is undergoing development and maintenance.

5.2 Integrated Architecture

Chapter 3 presented two axes along which architectures can be evaluated: how centralized the system is and how hierarchical a system is.

A successful autonomous cross country navigation such as the one used as an example system in this thesis must combine many different knowledge sources such as road followers, obstacle avoidance systems, and strategic route planners to navigate a vehicle through unknown, unpredictable, and unaltered environments outside of laboratories.

In this task, total centralization is not feasible. The success of the task depends on quickly reacting to the environment, and centralizing all of the vast amount of information that flows through the different modalities would cause a crippling bottleneck. If there is to be any centralization, it must be centralization of information summaries. Since quick reaction to changes in the world is the key to the success of the task: the more that information can be summarized the better.

So why have any centralization at all? First, by centralizing information the system can combine the outputs of the various knowledge sources to come up with a single course of action. The information center also provides a standard rendezvous point and method for the various knowledge sources as well. Without standardization there remains the problem that subsumption architectures have that each new knowledge source and extension has to be hand crafted into the system with many possibilities for unforeseen interactions and side effects.

In order to achieve the desired balance of independence and centralization most of the sensory modules in our systems are defined as "behaviors." These behaviors are independent entities that use their specific algorithms and knowledge of the world to produce a judgement of the world. The judgements of the world, i.e., the modules' estimates of good and bad courses of action for

the vehicle to take, are combined by a centralized arbiter, which then takes the "best" course of action given all of the collected judgements.

How hierarchical can a system be that implements this task? The world in which the system acts is too unpredictable for a strictly hierarchical system to work efficiently. A highly detailed plan will quickly be rendered useless through contact with the unexplored and unpredictable real world. Plans for this system must be in the form of tendencies, goals, and constraints rather than detailed paths and courses of action. If a plan does not contain room for the behaviors to maneuver around and deal with the unexpected, then the plan will have to be discarded and reworked whenever the unexpected is encountered. The need for flexibility has to be balanced with the fact that the more room for the unexpected a plan contains, the less useful it is.

Planners in this system cannot expect the world to be predictable, but they also cannot expect the system to be predictable. The system consists of relatively independent entities outputting commands through an arbiter. This level of independence of the components means sacrificing some predictability even when the world obeys the predictions. A higher level system can attempt to channel these independent entities through affecting how they are arbitrated, but the higher level system cannot predict exactly what they are going to do in the face of the data from the world, so detailed "predictive control" of the system is difficult for a planner to do. In fact, the individual modules cannot predict how the system will use their output, and thus predictive control is not useful even for the individual modules of the system.

I will split the architecture into three levels for discussion purposes. There is the sensory-motor level, in which there are independent behaviors monitoring the world and producing judgements on how to act in the world. There is a mission level, which attempts to herd the modules of the sensory-motor level to toward the ultimate goals of the task. Finally there is an actuator level which actually implements the wishes of the sensory-motor level.

5.2.1 Sensory-motor level

The sensory-motor level is the heart of the system. The sensory-motor level allows the system to survive and function in unstructured, unknown, or dynamic environments. At any given time, many different sensory-motor modules will be active, each giving possibly conflicting com-

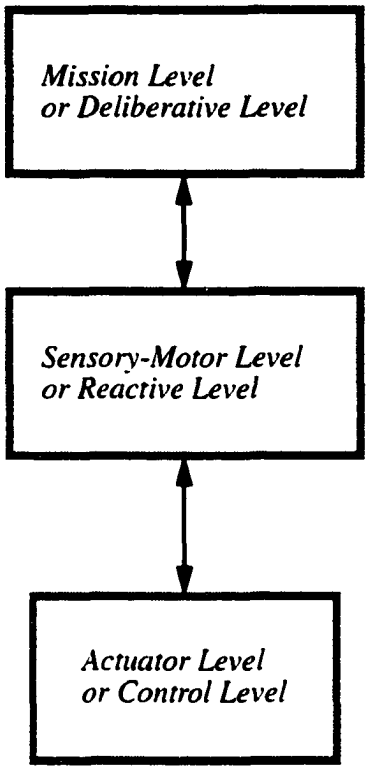


Figure 26 The integrated architecture

mands. To function effectively, we impose an architectural framework on these sensory-motor modules to provide a structure with which the system may be developed, tested, debugged, and understood. This architectural framework for arbitrating between the commands of the multiple sensory-motor modules is the work of Rosenblatt and is called the Distributed Architecture for Mobile Navigation (DAMN)[Langer et al., 1994]. As with other arbitration schemes [Payton, 1986, Arkin, 1989], under DAMN multiple modules concurrently share control of the robot. In order to achieve this, a common interface is established so that modules can communicate their intentions without regard for the origin of those intentions.

A scheme is used where each module votes for or against various alternatives in the command space based on geometric reasoning; this is at a higher level than direct actuator control, but lower than symbolic reasoning. This reasoning at the geometric level is crucial to the successful operation of a robotic system in the real world, and yet it is not well understood.

Figure 27 shows the organization of the DAMN architecture, in which individual behaviors such as road following or obstacle avoidance send votes to the command arbitration module;

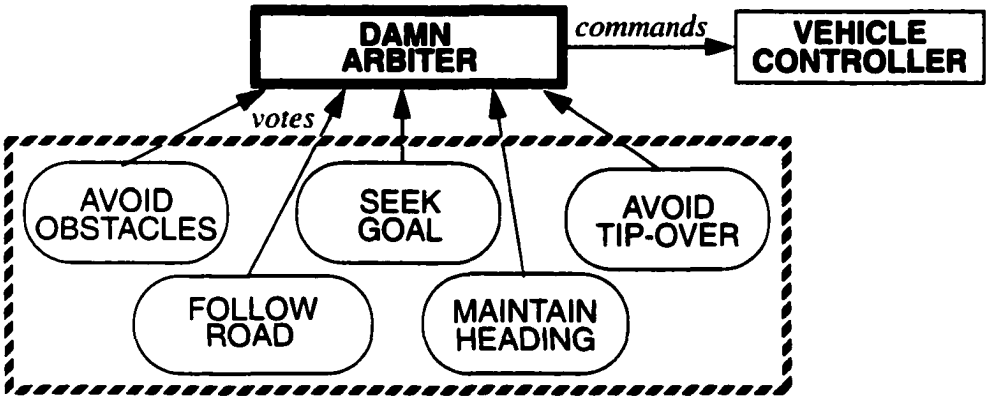


Figure 27 Behaviors sending votes to arbiter

these inputs are combined and the resulting command is sent to the vehicle controller. Each sensory-motor module, or *behavior*, is responsible for a particular aspect of vehicle control or for achieving some particular task; it operates asynchronously and in parallel with other behaviors, sending its outputs to the arbiter at whatever rate is appropriate for that particular function. Each behavior is assigned a weight reflecting its relative priority in controlling the vehicle.

The DAMN architecture is designed to integrate the basic capabilities essential to any mobile robot system, i.e., safety behaviors which limit turn and speed to avoid vehicle tip-over or wheel slippage, obstacle avoidance behaviors to prevent collisions, as well as various auxiliary behaviors. As new functions are needed, additional behaviors can be added to the system without any need for modification to the previously included behaviors, thus preserving their established functionality.

DAMN is designed so that behaviors can issue votes at any rate; for example, one behavior may operate reflexively at 10 Hz, another may maintain some local information and operate at 1 Hz, while yet another module may plan optimal paths in a global map and issue votes at a rate of 0.1 Hz.

In a distributed architecture, it is necessary to decide which behaviors should be controlling the vehicle at any given time. In some architectures, this is achieved by having priorities assigned to each behavior; of all the behaviors issuing commands, the one with the highest priority is in control and the rest are ignored [Brooks, 1986, Rosenschein and Kaelbling, 1986]. In order to allow multiple considerations to affect vehicle actions concurrently, DAMN instead uses a

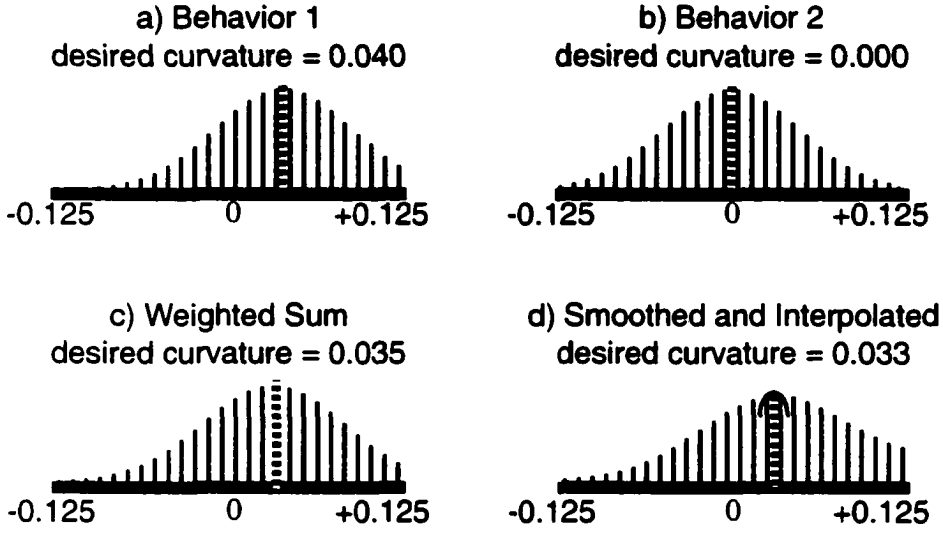


Figure 28 Command arbitration

scheme where each behavior votes for or against each of a set of possible vehicle actions [Rosenblatt and Payton, 1989]. The set of vehicle actions are defined a priori by the designer of the arbiter and must be fixed and consistent for all behaviors. An arbiter can then perform *command fusion* to select the most appropriate action. While all votes must pass through the command arbiter before an action is taken, the function provided by the arbiter is fairly simple and does not represent the centralized bottleneck of more traditional systems.

In the case of the turn arbiter, each behavior generates a vote between -1 and +1 for every possible steering command, with negative votes being against and positive votes for a particular command option. The votes generated by each behavior are only recommendations to the arbiter. The arbiter computes a weighted sum of the votes for each steering command, with the weights reflecting the relative priorities of the behaviors. The steering command with the highest vote is sent to the vehicle controller.

The arbiter collects the new votes from each behavior that has sent them, and performs a normalized weighted sum to find the turn command with the maximum vote value. In order to avoid problems with discretization such as biasing and "bang-bang" control, the arbiter performs interpolation. This is done by first convolving the votes with a Gaussian mask to smooth the values and then selecting the command option with the highest resulting value. A parabola is then fit to that value and the ones on either side, and the peak of the parabola is used as the command to be

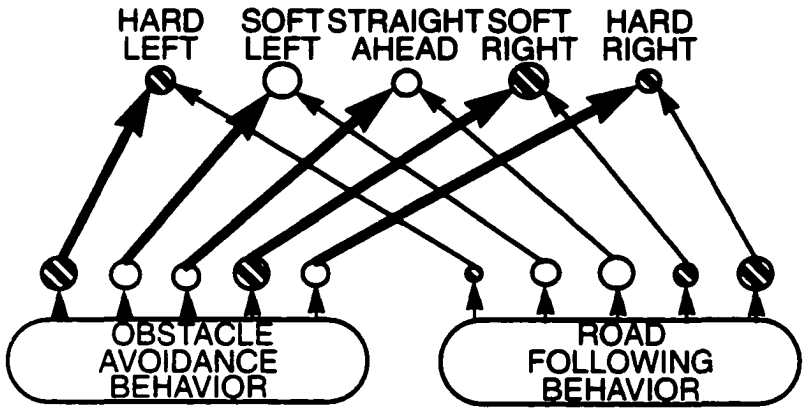


Figure 29 Command fusion in DAMN

issued to the controller. This process is illustrated in Figure 28, where the votes from two behaviors (a & b) are linearly combined (c), and then smoothed and interpolated to produce the resulting command (d).

An entirely separate speed arbiter with its own set of associated behaviors is used to control the vehicle speed. Thus, the turn behaviors can vote for turn commands without concern that the absolute magnitude of their votes will affect vehicle speed. At present each speed behavior votes for the largest speed possible which meets that behavior's constraints, and the arbiter simply chooses the minimum of those maxima, so that all speed constraints are satisfied.

As a simple example to illustrate the manner in which votes are issued and arbitrated within DAMN, consider the case in Figure 29 where two behaviors are active, one responsible for obstacle avoidance and the other for road following (only five turn options are shown for simplicity). The magnitude of a vote is indicated by the size of a circle, with a large unfilled circle representing a vote of +1, a large striped circle a value of -1, and a small circle a value near 0. Thus, the road following behavior is voting most strongly in favor proceeding straight and less favorably for a soft left turn, and voting against hard left or any right turns; the obstacle avoidance behavior is voting against a hard left or soft right, and allowing the other turns as acceptable, with soft left being the most favorable.

Given a task specification, a system integrator assigns appropriate weights to the outputs of the different behaviors. In the current automated army scout task, avoiding obstacles is more important than staying on the road, so the obstacle avoidance behavior is assigned a higher weight than the road following behavior, as indicated by the thicker arrows in the diagram. The arbiter

then computes a weighted sum of the votes it has received from each behavior, and the command choice with the highest value is selected and issued to the vehicle controller. In this case a soft left turn would be executed, since its weighted sum is the greatest, thus avoiding any obstacles while still more or less moving down the road.

The behaviors vary wildly. Some, such as neural net road followers have no internal state; some, such as local obstacle avoidance systems, have local state; and some, such as goal seekers, have global state. The cycle times for the behaviors vary from 10 Hz to 0.1 Hz. Even the language in which the behaviors are written will vary across the different behaviors. The only thing that the behaviors have in common is they all output command votes to the DAMN arbiters.

The DAMN arbiters for curvature and speed are examples of how centralization is used, but minimized. The turn arbiter is a centralization point, but only a centralization point for commanded turn radii. Using command fusion allows us to minimize the amount of information that has to be centralized and "reasoned" about. With command fusion there is no real intelligence bottleneck as there would be if each behavior had to present reasons why it generated the motor commands it did and then the arbiter had to merge all of the information from all of the modules to make a decision. The minimization of bottlenecks maximizes the system throughput and minimizes the latency between sensing and action as well as providing a consistent and simple structure for adding new sources of knowledge. The price we pay for this consistent and simple command fusion is that it can be dead wrong. Reasoning about the results of sensor fusion might identify more global aspects of a situation that argue for commands that cannot be extracted simply by combining disparate commands. Given our current task and our current sensor capabilities, this kind of error does not occur. Moreover, the reconfigurable interfaces give us a path for transition to a different type of architecture if it becomes necessary. Such a transition would be painful, but the reconfigurable interfaces would maximize the reuse of the sensory-motor modules designed for use with DAMN.

5.2.2 The mission level

The sensory-motor level is where the immediate, tactical decisions of moving the mobile robot are made. The mission level is where the strategic long range decisions are made. The sensory-motor level is primarily concerned with sensing the environment and reacting, while the

mission level is where deliberation and interpretation of meanings and events is done. Another way to think of it is that the mission level thinks globally while the sensory-motor level thinks locally.

The mission level primarily affects and directs the sensory-motor level by choosing which behaviors are active and how their outputs are weighted. The mission level also initializes the individual behaviors with parameters and modes that affect their operation. Modules on the sensory-motor level sense the world and produce information about it. In the current system, the reconfigurable interfaces convert this information into command votes for the DAMN arbiter and status variables that can be accessed by the mission level components. If necessary, reconfigurable interfaces could be designed to give mission level components access to the semantic information rather than simply converting it into command votes as is currently done.

The mission level is almost useless for much of the life of the system. Most of the testing will be done with a fixed set of behaviors with an empirically discovered set of weights and parameters, collectively known as a mode. Each mode will be thoroughly tested on its own to tune the parameters and weights and find errors due to unforeseen interactions. The mission level is needed when the system needs to execute missions which involve autonomous sequencing of different modes, monitoring of system progress, and handling of errors.

Chapters 6 and 7 go into detail about the tools we have developed for use in the mission level. These tools further reflect the philosophy of emergent architectures in that they have been developed specifically to meet the current needs of our tasks rather than some possible future requirements.

5.2.3 Actuator level

The actuator level can be thought of as the vehicle controller. The vehicle controller provides the means for determining where the vehicle is and what state it is in and the means for moving the vehicle. More details on this system can be found in [Amidi, 1990].

Most of the modules in our robot systems, whether they are in the sensory-motor level or the mission level, need to know where the vehicle is and what state the vehicle is in, i.e., how fast is it

moving, where is it steering, etc. A more select group also needs to command the vehicle to actually move, i.e., to steer along an arc at a given speed. All of these modules are considered external clients by the controller, and connect to the controller directly via TCP/IP links.

The actuator level is implemented as a group of processes running under a real time operating system communicating through shared memories. The processes on the actuator level combine information from odometry, actuator encoders, gyroscopes, and inertial sensors to keep a relatively accurate estimate of the vehicle position for the external clients. Other processes monitor the vehicle state, and try to keep the vehicle safe as the actuator implements the commands.

The actuator level is implemented differently than the sensory-motor and mission levels. The more intense real-time issues demand that the vehicle controller resides on a board running a real-time operating system and its individual components communicate using explicit shared memory rather than the architectural infrastructure that is used by the rest of the system. The actuator level is qualitatively different than the rest of the system, and is implemented as such.

5.2.4 Comparison with other architectures.

Layered architectures which have a control level, a reactive level, and a deliberative level are a common solution to building a system that can operate reliably in an unpredictable world. Architectures such as ATLANTIS[Gat, 1992] or GLAIR [Hexmoor et al., 1993] both recognize that each level has different requirements and thus each level should have its own structure and data flow.

The difference between these types of architectures and our integrated architecture for cross country navigation is that other layered, heterogenous architectures have a fairly strict separation of the levels, i.e., communication between the levels is often much more restricted and more structured than communications within a level. An example of this kind of strict separation in the integrated architecture is in the relationship between the actuator level and the rest of the system. The processes in the actuator level are defined by the very fact that they exist in that layer. They communicate freely with each other, with different processes controlling the velocity and speed of the vehicle while monitoring and integrating sensor information such as odometry and positions sensed through global positioning satellites. Modules in the other layers only interact with the

actuator level through accessing vehicle state and position and through issuing actuator commands, not by interacting with the individual processes of the actuator level.

The mission level and the sensory-motor level, on the other hand, are not strictly separated. Modules in these levels are not defined by being in the levels. The levels represent a taxonomy for convenience of description rather representing any physical, computational, or communication structure. Modules in the mission level will tend to use tools that are more deliberative and global while modules in the sensory-motor level will tend to be more reactive and local, but there are modules that blur the lines.

An example of a level bending module is the D* behavior. D* is a global trajectory planner and replanner [Stentz, 1994]. D* constantly monitors where the vehicle is, and constantly updates its internal global cost map with obstacles that the obstacle avoidance system has detected. The D* algorithm (or Dynamic A*) is functionally equivalent to an A* replanner but is over 200 times faster than just replanning the entire trajectory with A* in reaction to new information or unexpected vehicle positions.

D* is placed at the sensory-motor level because it outputs command votes to the DAMN arbiter. D* does not fit into the typical slot for a sensory-motor module because it is constantly deliberating about actions that have global, not local, impact. D* can straddle the line between the mission level and the sensory-motor level because it is an imaginary line. Modules are not defined by what level they are in, but rather by what tools they use and what other components they connect to. Thus, there is no problem with D* existing simultaneously in the mission level and the sensory motor level, it simply uses the resources of both.

5.3 Road following architecture

The road following task is much simpler than the integrated architecture task, and thus the architecture is much simpler. The purpose of the road following task is to follow roads as well as possible. There is a single modality with a single purpose. In the pure road following task, all information flowing in the system has to do with following the road, and therefore all of that information is centralized through the road following algorithm. This centralization is necessary for the task to succeed.

Because the system is simpler, there is more room for predictive control and hierarchy. The road follower will know that it is the only component in the system producing steering commands, and that any controller will simply attempt to execute the steering commands as precisely as possible. This simplicity, coupled with the inherent structure of roads does allow for predictive models to be used to make the control of the vehicle more precise. Unfortunately, if our road following system does take advantage of the predictability of the task, the algorithm will not transport easily into the more unstructured integrated architecture task. We believe that the increased precision is not worth the inability to use the algorithm in other systems. Having a vehicle that stays within 1cm of the center of a lane rather than 10cm is not worth having two very different road following algorithms for the road following task and the integrated task.

"Higher level" planners are mostly irrelevant in the development of the road following system. As with the integrated task, high level planners will not be feeding trajectories to the road following system, but rather commands for how to follow whatever road the system is presented with, i.e., where to drive in a lane, when to change lanes, etc. Because the system is simple, with one road following algorithm controlling all of the steering, a high level system could model how it believes the "lower" level system will react to commands. In reality, in most cases for this task, the high level system can be replaced with a simple user interface.

Figure 30 shows the structure of our road following system. The single purpose of the task allows the system to collapse down to a single process. Instead of having external processes grab images, maintain positions, and control actuators, all is done inside of one process. The road following algorithm digitizes an image, processes it, and then uses the steering control subsystem to actually move the steering wheel. Meanwhile, the position maintenance system can be used to track windows in the images if necessary. The limited scope of the task means that the support operations, such as position maintenance and steering control, are limited enough to be incorporated into a single process with the road following algorithm. The system can use the knowledge of exactly what is needed by the single controlling algorithm to take advantage of smart hardware, such as PID control boards or Global Positioning Satellite systems. Thus the system provides only what is needed to make the road follower work with a minimum of overhead. This architecture is tuned to the task it is built for, rather than being placed in any generic multipurpose architecture.

The same system works with both a neural network road follower, ALVINN [Pomerleau, 1989], and an adaptive feature based algorithm, RALPH[Pomerleau and Jochem, 1996].

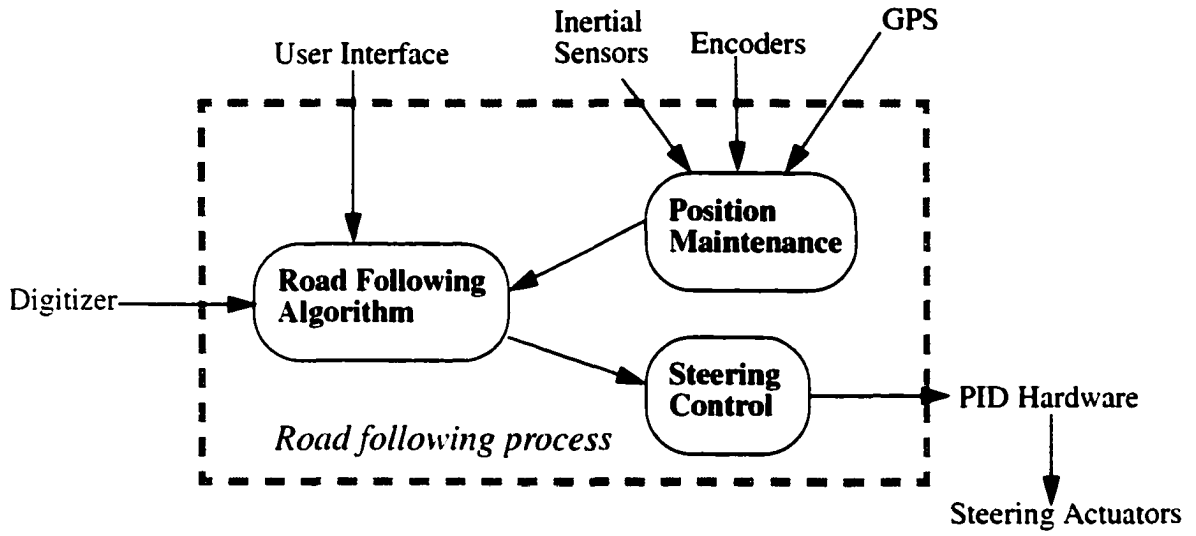


Figure 30 Road following architecture

The road following algorithms developed at Carnegie Mellon were developed primarily with this architecture and focused task in mind, and their developers continue to work primarily within such an architecture. Reconfigurable interfaces allow a system integrator to extract the road following algorithm in its latest state and place it as a component of the integrated cross country system.

5.4 Moving modules from architecture to architecture

A key goal of the emergent architecture approach is to free architectures to change with the needs of the task. If code is easy to move between two different projects with very different sets of requirements, then code will be easy to carry through incremental changes in architectures resulting from incremental changes in requirements and ambitions. Thus, system developers can change architectures and experiment with new ones as the system tasks evolve while still seamlessly carrying old code forward in the process.

There are limits to this process. For example, if a road following system uses predictive control, its outputs are not just based on the current inputs and past history, but on a model of the systems outputs on the world extrapolated into the future as well. Such a predictive road following

algorithm might well work on its own, but would not work well integrated as just one behavior of many in a multi-modal arbitrated system. In such a system there is little chance that what the road follower believes will happen with its output will, in fact, happen, so any predictive control will be at best useless and at worst harmful.

I use ALVINN, our neural network based road follower [Pomerleau, 1989], as the example algorithm for seamless transfer between two different architectures: the integrated, multi-modal architecture and the dedicated road following algorithm. ALVINN is well suited to this since it is not predictive at all. ALVINN detects the road, and issues commands to steer the vehicle onto the road based on its trained association between road images and steering commands.

Figure 31 shows the source and destination interfaces that the ALVINN algorithm uses. ALVINN gets images from a digitizer, and uses its trained neural nets to determine where the appropriate road or lane is and outputs the arc which will best bring the vehicle onto that road and a set of statuses including how confident it is in that decision.The configuration source provides passive information, such as vehicle parameters, and active information such as when to stop and start the algorithm and when to change lanes. The vehicle position and state are used for such tasks as performing lane changes or training neural nets.

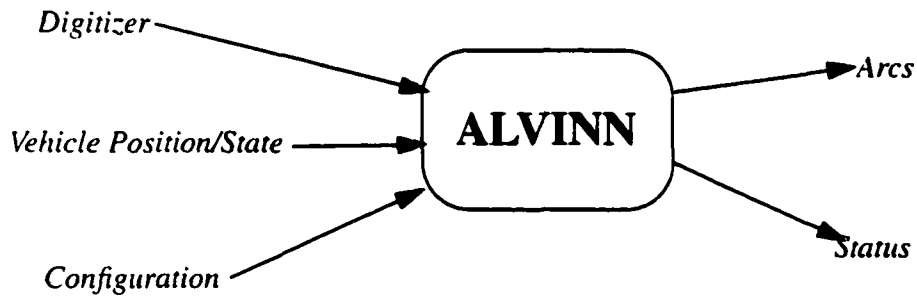


Figure 31 Sources and destination interfaces for ALVINN

In both the road following architecture and the integrated architecture, ALVINN can use the same interface configurations for all of the source interfaces. The digitizer will remain the same, the local position source will remain the same, and the configuration will be done by the same sets of files and external commands. The real difference is in the destination interfaces. In the stand-alone architecture (see Figure 30), the status is ignored and the arc goes straight to a vehicle con-

troller. In the integrated architecture, the status is published to high level plan monitors and the arc gets converted into a set of arc votes and sent to an arbiter (see Figure 26).

The reconfigurable interfaces allow this module to transition between the two architectures running on the same vehicle simply by changing two configuration strings, one for the arc output and one for the status output. At present, these configuration strings exist in option files that are read when ALVINN starts up. ALVINN reads these strings at run time and uses the reconfigurable interface tools to create the interfaces.

In the road following architecture, the status configuration string is simply,

```
"none"
```

which means that the output of status is not done to any destination. In this same architecture, the arc configuration string is,

```
"internal"
```

to indicate that the arcs will go straight to a simplified vehicle controller which is internal to the road following process. In the integrated architecture, the status configuration string is,

```
"ipt"
```

With the status interface set up with this string, the algorithm will use IPT to publish its status to whatever external module requires it. In the integrated architecture, ALVINN's arc configuration string is

```
"damn"
```

to indicate that the single arc produced by the algorithm will be converted into a set of arc votes. The arc votes will be a gaussian with the peak at the desired arc. The interface then sends the arc votes to the DAMN steering arbiter.

So, just by changing two configuration strings in a configuration file I can move a highly complex sensory-motor module from one architecture and task to another. The different strings simply switch between which of the possible interfaces in a library is used, so there does not even need to be any recompiling or relinking of the module. Thus, ALVINN algorithm developers only need to worry about one version of the ALVINN module, which can be inserted in a variety of different

architectures rather than having a development version of ALVINN which must be painfully modified every time an integrated test is desired.

5.5 Limitations of reconfigurable interfaces

Reconfigurable interfaces are not a panacea which allows us to put a module in any arbitrary architecture: They first of all allow a module to move easily between a given set of architectures, and second provide a consistent path of migration for moving to a new, currently incompatible architecture.

For example, the two architectures presented here for the integrated cross country system and the dedicated road following system, both make the same assumption about ALVINN: that ALVINN processes quickly enough that its output is a good estimate for where the road is at the time the output is received. In other words, when the DAMN arbiter in the integrated cross country system receives votes for where to drive from ALVINN or when the steering control in the road following system receives a command from ALVINN, both systems ignore the latency between the time of sensing and the receipt of the information. They both assume that the input from ALVINN represents the current state of the world, even though that input actually represents the state of the world roughly 30 milliseconds earlier.

It is easy to imagine an architecture which would not accept this assumption, and which would demand that ALVINN pair its outputs with a time value representing exactly, down to the millisecond, when the data that produced those outputs was acquired. ALVINN in its current state with the current reconfigurable interfaces could not just be put into this new architecture. In order to move ALVINN to such an architecture three steps have to be taken,

1. ALVINN has to be changed to record exactly when it processes its images.
2. The reconfigurable interfaces have to be changed to accept the new time value from ALVINN.
3. The new ALVINN has to be integrated into the new system.

Although reconfigurable interfaces do not provide an instant solution for placing ALVINN in the new architecture, they do provide a mechanism for making the three steps occur in parallel

rather than strictly in sequence. For example, while the ALVINN developer is modifying ALVINN to incorporate time stamping its processing, the system integrator can introduce a temporary reconfigurable interface which takes the output of the old version of ALVINN and estimates the time at which the image that produced the output was taken given an empirically discovered average latency time. The system integrator could then start working on many of the issues of integrating ALVINN into the new system using this approximated output while the ALVINN developer is coming up with the final solution.

5.6 Developing a module with reconfigurable interfaces

Having the ability to easily migrate a module from architecture to architecture does not just help concurrent development of projects, but also is a boon for the development and integration of an algorithm for a single project.

The example I will use is the D* strategic route planning behavior [Stentz, 1994]. In an integrated system, D*'s behavior is highly dependent on many different sub-systems. In order for D* to be tested there needs to be working obstacle detection as a source of new information, working mission planning to tell D* where to go and when to start and stop, a working arbiter to react to D*'s outputs, and a working global positioning system to accurately and repeatably tell D* where it is. Testing D* in our current integrated system requires running and properly operating nine separate processes. If there are problems with any of these processes, there will be problems with D*.

The development, debugging, and integration of D* with and without reconfigurable interfaces shows how reconfigurable interfaces smooth the integration and testing process. Without reconfigurable interfaces, D* will be initially developed as a totally stand alone module. The developer will only have one process to run and test, and the module developer will hand craft simulated information for testing and debugging. Once the D* module is ready to integrate into the larger system, the module developer will sit down with the system developer and together they will rip through the D* module and modify it in order to process real data and produce real outputs in the current version of the system.

Usually, this process results in a version of D* that the module developer works on and improves and publishes papers on, and a separate version which is "handed off" to the system developer. If there is reconfigurability in the module, then that is usually hand crafted by the module developer through a maze of conditional statements that permeate the module. Even if the module developer does the best job possible at isolating the architecture dependent code in a small portion of the module, the system integrator still has to deal with the module developers particular interpretation of reconfigurability. The problem remains that every time the system architecture changes, the system integrator has to go into each module and rewrite the "system" options, and each rewrite will be slightly different since they were all written by different people. Integration errors and side effects become inevitable.

In my experience, reconfigurable interfaces provide a separation between the system integrator and the D* developer. The D* developer works on the algorithm, while the system integrator works on providing the D* developer with the information that is needed and the outputs that are required. The D* developer does not have to be intimately acquainted with the system architecture, which might change, and the system developer does not have to be intimately acquainted with the algorithm, which is undergoing continual research. Each developer remains in his area of specialty, thus increasing the efficiency of the integration effort.

Reconfigurable interfaces not only increase the efficiency of integrating D* into a system, properly used they also increase the efficiency of isolating and fixing bugs that come up in the process of testing the integrated system. For example, if the system developer guesses that there is a bug in the system due to a fault in the D* module, the system developer can use the reconfigurable interfaces debugging classes to carefully log and monitor the inputs and outputs of D*. If the system developer still believes the fault to lie in D*, then the integrator can attempt to reproduce the error using simulated or previously logged inputs. Eventually the system integrator should be able to produce a set of configuration strings and data files that reproduce the bug in a stand alone fashion, so only the D* module has to be run with the rest of the inputs coming from passive sources such as internal simulators or logged data. Once the system integrator has reached this point, then this set of configuration strings and data can be passed off to the module developer who can find and fix the bug while only dealing with that one module.

This approach is vastly more efficient than the alternative debugging procedure without reconfigurable interfaces. Without the reconfigurable interfaces, bugs that are found in the operation of an integrated system tend not to be reproducible without the whole system running. When debugging a problem that shows up in a real system run, the choices are either that the system integrator becomes an expert on D* or that the D* developer becomes an expert on the current system. Bugs that require the proper running of a dozen modules to reproduce are much harder to find than bugs that require the proper running of a single module. When a system developer is dealing with a team of module developers, it becomes very difficult to bring each one in whenever there is a problem with their module. This problem becomes greatly exacerbated when the system integrator is hundreds of miles distant from the module developer. Reconfigurable interfaces provide a consistent set of methods and locations for isolating and reproducing the bugs that arise in the course of system development. This bugs can be then either isolated in problems with individual modules to be reproduced and debugged simply by the module developers or they may be determined to come from interactions between modules, in which case the problem should be dealt with by the system developer. Thus reconfigurable interfaces make the complicated job of debugging a large mobile robot system easier, especially when the integrator and system developers are physically separated by long distances.

Where do the reconfigurable interfaces for D* come from? They come from primarily from a list of what D* needs and what D* produces. D* needs

- A position source, to tell D* where it is.
- A terrain information source, for what terrain is traversable and intraversable.
- A source for when to start and stop, where to go, and how to get there, which I call a configuration source.

D* produces

- A destination for arc votes to drive the vehicle.
- A destination for status variables such as how D* is doing and the current cost to the goal.

Most of these sources and destinations already exist for other modules, since there are fairly common interface needs across a system. Even if there is not an exact match for a source or destination, I can usually define a subclass of an existing interface which satisfies the needs. For example, there will be no existing reconfigurable interface which satisfies all of the needs of a D* configuration source, since probably only D* will need configuration parameters which specify costs and weights for a D* search, but there will be a configuration source which specifies starting and stopping. A subclass of this general configuration source will inherit these general methods, and I can then add the methods for getting D* specific parameters to extend the interface to satisfy D*'s requirements. By using inheritance I can save coding time and provide consistent interfaces for general parameters such as module starting and stopping.

So the reconfigurable interfaces provide a systematic way to isolate and separate the jobs of the system developer from the D* developer. The system developer is freed to chase down bugs that arise from interfaces and interactions between modules and the D* developer is free to chase down bugs that arise from problems with the D* module.

5.7 Conclusions

The architectures presented in this chapter are snap shots. I do not claim that they are the best architectures for cross country navigation and road following. They are simply architectures that successfully meet our current needs for cross country navigation and road following. The integrated architecture has been used both at Carnegie Mellon and at Lockheed Martin to pull together systems using road following, teleoperation, obstacle avoidance and strategic goal seeking to traverse kilometers of unstructured terrain. The road following architecture has driven across the United States with the system in active control of the steering wheel for 98.2% of the trip. As successful as these architectures are, as we discover new information, invent new techniques, and expand our ambitions, the architectures must change.

For example, experiments with the current integrated architecture have shown that our command fusion approach for integrating multiple sensory-motor behaviors may throw away too much information, i.e., not enough information is centralized at the arbiter for it to make correct decisions. We are experimenting with changing the arbitration paradigm to use utility maps rather

than actuator commands, i.e., have the sensory-motor modules report to the arbiters which areas are good or bad to drive in rather than reporting a set of arc votes. The expanded arbiter can use the extra information to both eliminate some ambiguities that exist under the current architecture and to exercise predictive control to take into account vehicle dynamics at higher speeds. An increase in the computational abilities of the system may offset the additional cost of centralizing and processing more information in the arbiter.

As another example, the road following architecture is a wonderful single purpose system, but we will need to integrate it with other components for speed control in addition to steering control. Interaction with obstacle avoidance, convoying modules, and strategic decision makers will require a different software architecture.

The emergent architecture technique acknowledges that architectures cannot be treated as write once/read only systems: They must be allowed to change as tasks and requirements change. The approach provides tools, i.e., reconfigurable interfaces, which ease the transition to new architectural paradigms, such as going from a command fusion based method of integration to a utility map based method of integration, or going from a stand alone road following system to a full highway navigation system. The same reconfigurable interfaces that allow a road following component to move from one project to another can allow a road following component to be moved from one architectural paradigm to another within the same project with the least possible impact.

Finally, the tools of emergent architecture not only assist in moving and reusing components across systems, they assist in the development and integration of individual components for individual systems. In a traditional approach, the "bandwidth" of interaction between a component developer and a system developer is huge. If the system developer discovers bugs or shortcomings in a component in the process of integration, the only two possibilities for fixing them was for the system developer to learn everything there is to know about the component, or for the component developer to come face to face with the system developer to interact with the system in order to recreate, isolate, and finally, repair the problem. With an emergent architecture's reconfigurable interfaces the "bandwidth" of interaction between system developer and component developer has been largely reduced to e-mail from a system developer who sends the configura-

tion strings that duplicate the problem in isolation, and then code fixes from the component developer which fixed the problem.

Chapter 6 Plans in a Perception Centered System

6.1 Who needs plans?

The systems that I have presented are perception centered. In both the integrated cross-country task and the road following task the vast majority of computational, developmental, and logistical resources are taken up by perceiving and reacting to the world rather than deliberating and sequencing the actions.

Although the systems are perception heavy, they still require some sort of plan execution and monitoring system to perform complex missions, i.e. there needs to be some modules and infrastructures for sequencing modes of operation appropriately. For example, a road following system does fine at following a road, but by itself it cannot decide which way to go at an intersection, or even how to slow to a stop at an intersection before traversing it. Similarly, any one "mode" of the integrated system, such as teleoperation or cross-country navigation, will keep the vehicle alive, but the system needs to be able to transition between modes in order to satisfy the overall objectives of the mission. Questions such as which mode to enter next, how to get to that mode, and what parameters that mode needs, can only be answered by a plan execution and monitoring system.

Another motivation for using plans is that perception for outdoor mobile robots often fails. The more information and heuristics that a plan contains that can help perception subsystems do their jobs, the more likely it is that the task will be successfully performed. Plans can also be used to attempt to recover from the inevitable failures of perception elegantly instead of just grinding the whole system to a halt.

The role of plan execution and plan monitoring must be put into context. The goal of my systems is not to produce a general, intelligent, independent, autonomous robot which as a side effect performs some interesting tasks. The goal of my systems is to perform those tasks, usually under the close supervision of a human. The plan execution and monitoring tools should be considered as system engineering tools which support the "important" work of the system: perceiving the salient features of the world and reacting to them appropriately to perform the system's task.

Within this context, the plan execution and monitoring tools have two conflicting requirements:

- **Predictability**: The plan specification and execution must be straightforward, since most of the testing of the system will be simple sequences of actions to test and tune the perception and reaction subsystems. The fewer unnecessary features and ambiguities in the specification and execution of a plan, the fewer bugs will originate from the plan execution and monitoring system, and thus, the more time can be devoted to eliminating problems in the perception and reaction subsystems.
- **Flexibility**: Although most of the testing of the system will be simple sequences of actions, the actual task may involve detecting errors and reacting to contingencies. The more that the plan monitoring and execution system can do on its own, the less burden there will be on an external human operator.

In the tasks I am addressing, the emphasis on developing reliable perception and reaction subsystems means that our plan execution and monitoring systems put more weight on the predictability constraint rather than the flexibility constraint. I have built a system that walks the line between the two for us: SAUSAGES, the System for AUtonomous Specification, Acquisition, Generation, and Execution of Schemata[Gowdy, 1994]. SAUSAGES lets a system integrator

quickly prototype and debug scenarios while having the flexibility and power to address a wide variety of tasks.

SAUSAGES is not a reasoning system, so if a task demands that level of sophistication, SAUSAGES by itself it would not be appropriate, but it is appropriate for the kinds of tasks for which we use the integrated architecture.

6.2 Plans in the integrated architecture

The plans in the integrated architecture have two purposes: To direct modules, and to help modules. The plans will select what modules work on a given stretch of the plan, and serve to support and guide the modules of the sensory-motor level to achieve the ultimate goals of the task.

6.2.1 Plans as module selection

The system needs to be directed through stages, such as following a road, then traversing open country, then stopping and performing reconnaissance.

Plans such as this can be considered as series of modes. Each mode represents a set of modules that will be active and how they communicate with their resource processes. For example, one mode might be goal seeking cross country. In this mode the system would need D* and a local obstacle avoider active with, for instance, empirically chosen behavior weights of 0.1 and 0.9 respectively. The plan would have to start up each of the behaviors properly and send these weights to the DAMN turn arbiter. Modes might also use non-sensory-motor perception modules, i.e., modules that perceive events that do not directly translate to motor commands. These events include detecting intersections, tracking other vehicles, recognizing signs, etc.

The plan execution system also has to be concerned with all of the details involved in the transition between modes. The planning system must detect failures in transitions, i.e., that modules are not ready to start or have failed, and must then deal with those problems.

The plan execution system must also detect when the system is not acting to fulfill the mission goals. The modes that the mission level sends to the sensory-motor level do not guarantee the correct actions. The sensory-motor level is constantly working first to keep the vehicle safe, then to

nudge the vehicle towards its goals. The plan execution system must recognize when the vehicle is not making satisfactory progress, and take action, whether through human intervention or autonomous replanning.

6.2.2 Plans as passive advice

Mission level plans for an outdoor mobile robot cannot simply be programs, since the task requirements demand that perceptual modules act as relatively independent agents. Agre and Chapman present a better metaphor that fits our requirements: plans as communications [Agre and Chapman, 1990]. In this approach, plans are not the center of the system. A planner communicates with other "entities" just as a human would communicate directions to another. The details are not filled in, allowing the other entity to improvise within the plan. Thus the planner is not the "top" object in the system, it is just another agent which guides, but does not dictate, the actions of the other agents in the system.

A better metaphor is plans as advice rather than just communications. A plan should not just contain the bare minimum of how to perform a mission, it should contain as much information as the planner can load into it. This extra information is not commands, it is advice based on the planner's best knowledge about the world. The plan is intended to help perception modules do their job, not to dictate how they should do their job. If it is good advice, the perception modules will perform better and the system will perform better. If it is bad advice, i.e., it contradicts the evidence sensed in the real world, or leads to failure of the perception module, since it is just advice the perception modules can then ignore it and complain to the agent that generated it so that advice will be better the next time.

This different metaphor for plans is more of a refinement of Agre and Chapman's views than a replacement. As an example, if someone has a map of the terrain between a subway station and an apartment and can communicate it, why not send the whole map annotated with the directions and visual aids rather than boil the knowledge down to a terse, relatively ambiguous set of verbal directions. The receiver of the map is still free to treat it as simply communications, and the successful execution of the "mission" is still determined by the interaction between the plan, the world, and the executor's skills. The difference is that now the system is better equipped to show up for dinner on time.

6.3 Options

The mission level of the integrated architecture must select the modes for the sensory-motor level, manage the transitions between modes, and monitor for errors. The mission level can also provide advice to the perception modules on where to look and what to look for given the current situation and goals.

6.3.1 Strategies vs. programs

Two major approaches to behavior specification for robots are using strategies and using programs[Agre and Chapman, 1990]. A strategic system lets the user define goals, sub-goals, and policies, and at any given point examines the current system state and performs a search across the policies to best move the system towards the final goals. A programming system produces a program, i.e. a set of task sequences and contingencies which lead the system to its goal (or to failure). The search through policies to achieve system goals is done by the programmer at the time of writing the program.

Strategic reasoning systems, such as the Procedural Reasoning System (PRS) [M. P. Georgeff, 1987] or the Reactive Action Packages (RAP) system [Firby, 1989], provide flexible, robust performance, especially when there are multiple methods available to attain a goal. They allow extremely flexible and compact specification of contingency behaviors, and have the possibility of responding to situations that were not explicitly expected: If an unexpected situation comes up, then the system may still reason its way to achieve some of the goals. In a programming system, if a contingency is not explicitly planned for before execution time, then the program will fail.

Clearly, if achieving and demonstrating high levels of autonomy is a driving purpose of a system, a strategic reasoning is a good choice. In other words, if the flexibility constraint is weighed very heavily, a strategic reasoning system should be the task manager.

A program is much more inflexible than a planning system, but it is more predictable. The very reasoning power of tools such as PRS and RAP imply that the goals and policies must be chosen very carefully in order to consistently elicit a given sequence of actions. In the integrated cross-country system the major point is to test and integrate perception modules in a semi-autono-

mous, supervised system, so a system integrator will want as little uncertainty as possible in sequencing and testing the various subsystems.

The distinction between programming systems and strategic planning systems can easily be blurred. Most strategic systems can be run using degenerate specifications in order to implement, essentially, a sequence of actions with simple, explicitly laid out contingencies. Similarly, since the strategic systems themselves are simply complicated programs, a program can be written in any language which is as capable of reasoning as any full blown reasoning system.

Although one tool could be used for the life of a project, the emergent architecture approach encourages the use of the right tool for the right task, and encourages switching those tools as needed. In a task such as the integrated cross-country system in which the perceptual subsystems are evolving, active research projects, it is important that the task management be as predictable as possible, even at the expense of flexibility, in order to reduce the number of bugs introduced from such "non-essential" areas. As the perceptual subsystems mature and stabilize, and as the tasks attempted involve more autonomy and strategic reaction, flexibility becomes much more important than stability, and strategic planners should be introduced as the foundation of the task management.

6.3.2 Types of programs

Most of the demonstrations which depend on the integrated architecture are simple, and not much is necessary to manage the tasks. For any one mission, a simple hard-coded program could be written that would do all the task sequencing and managing that is necessary. This would certainly satisfy the criteria that the plan execution system be as simple as possible.

Unfortunately, this extreme in simplicity ignores the flexibility constraint entirely. The problem is that the integrated architecture is not meant for just one fixed mission. The integrated architecture is the basis for a variety of related missions. Existing components will evolve as their programmers work on them, and thus the parameters and interplay between these existing components will need to be constantly updated and fine tuned to make a given mission work. Furthermore, new components will be developed which will enable new missions, which will in turn require experimentation and modification of the plans that the mission level produces.

Much of the work in robot programming has concentrated on complete control: with a single, specialized language, specify every aspect of a robot down to what real-time controller to use and how to interpret the various sensors. Such programming languages have been especially focused on programming manipulators and assembly work-cells, and have ranged from text programs with special constructs and libraries to specify and manage the many concurrent problems involved in such programming [VAL II, 1983, Taylor et al., 1983] to visual programming environments which let users point-and-click their way to a working manipulator or work-cell[M. W. Gertz, 1994]. Unfortunately, what our tasks need is not a complete specification of behavior from top to bottom, but rather a flexible specification of the task level behavior, with modules taking advice from the programming system and then using that advice to achieve the details of the behavior.

There are a variety of existing task-level plan execution systems. One problem with many of them is that they are embedded in an architecture which revolves around the plan execution system, and that system dictates how information flows in the system in order to make the task management easier. An example is the Task Control Architecture [Simmons, 1994], in which task control is the central issue in making the system work. Another example is the ESTEREL programming language [Berry and Gonthier, 1992], which makes correct execution of a formally verified "plan" the central issue in making a system work. These kinds of plan execution systems usually consider themselves architectures rather than components, and thus make too many assumptions about the nature of the system to be appropriate for my tasks. Although such plan execution systems can usually be extracted from their architectures to form the required "task control" components, at the time of developing the plan execution tools for the integrated cross country system they were not.

6.4 Description of SAUSAGES

SAUSAGES stands for System for AUtonomous Specification, Acquisition, Generation, and Execution of Schemata. SAUSAGES is designed to be a bridge between the worlds of planning and perception. It can be used to integrate perception modules together into useful systems with a minimum of planning work, and it is easy to use with external planners that can generate, monitor,

and adjust SAUSAGES plans to create intelligent and reactive behavior in the face of changing environments and goals.

A SAUSAGES plan is made of discrete semantic units called "links." A link can be thought of as a single step in a plan. A link has entrance actions that get invoked when the link starts up, and exit actions to clean up when the link exits. The link has a set of production rules associated with it that are added to a global production rule system. These production rules have actions that specify how to decide when the link is finished, how to tell when it has failed, side affects of the link, or whatever the system designer wants them to specify. These production rules are only in effect while the link is active.

Figure 32 A link of SAUSAGES

SAUSAGES uses a Link Manager process that manages the execution of links. It is the Link Manager that maintains the production rule system and starts and stops links. Most interaction with a SAUSAGES plan is done through the Link Manager by setting state variables that fire various production rules. The Link Manager also lets external modules remotely add links, run them, delete them, and even change them.

A SAUSAGES link also has a function that returns the next link to execute when it is finished. This is how a planner hooks together a series of links into a plan. This function can be as simple as returning another link, which might be part of a serial chain of links, or it could be conditional, where if one condition holds activate one link and if another holds activate another. This allows a plan writer to construct plans that are directed graphs of links rather than just a serial chain. Thus a planner can create a plan that has some intelligence and decision making built into it to handle common contingencies that might be foreseen.

Links can be composed of sub-links, to form hierarchical plans. This is a common abstraction in plan descriptions. A link which implements "go to airport", will decompose into several sub-links such as, "get in car," "drive to airport," and "park at airport." Each of these links could have

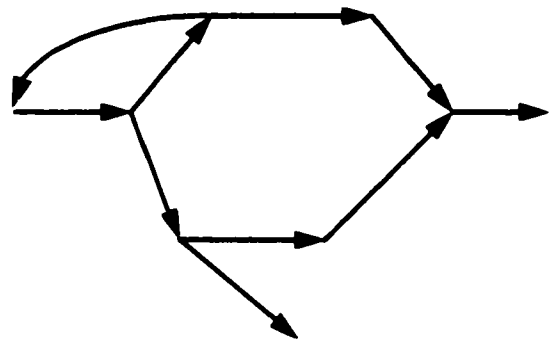


Figure 33 A plan as a graph of links

their own production rules with completion and failure conditions attached. The plan designer can specify that any failure be handled entirely at the level of the link that failed, or that failures get propagated up the link hierarchy.

SAUSAGES plans do not have to be "plans" in the traditional sense. They can simply be algorithms. An external planner can produce an algorithm for a situation, complete with looping or recursion, download it to SAUSAGES, and run it. We get this flexibility because the "next link" function is arbitrary, and instead of producing a predetermined link it could be a function that generates a new link based on the current link or any condition that is relevant. Thus SAUSAGES "algorithms" can be recursive and dynamic rather than simply iterative and static.

The plan graph can have more than one link active at a time. This way the Link Manager can simulate multiple threads of control. One common example for mobile robots is to have two threads of control, an action thread and a context thread. The action thread will control the robot motion and perception, while the context thread will monitor for global errors or major changes in the plan. The context thread can affect how the action thread transitions. The context thread can start and stop links in the action thread, or replace links in the action thread entirely.

So, SAUSAGES provides a planning language that fits my criteria for specifying and executing mobile robot plans. These plans can range in complexity from a simple linear linking together of different steps in a plan to actual algorithms including looping and recursion. The plans are predictable, because only active links effect the system. An advantage of using SAUSAGES rather than simply having a mission plan written in a standard compiled or interpreted language is that

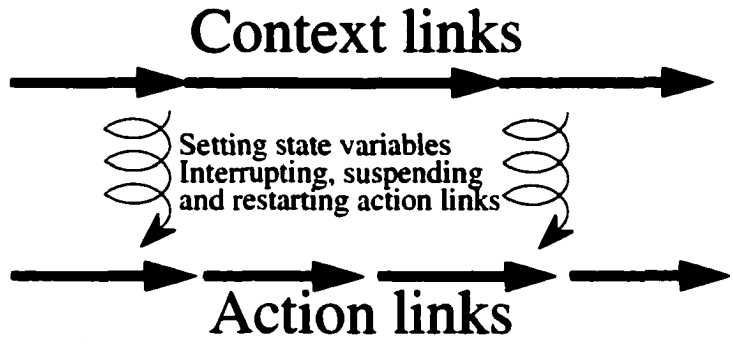


Figure 34 Concurrent links

SAUSAGES consistently composes plans out of semantically meaningful units which can be easily manipulated by external planners and monitors in useful ways.

6.5 Implementation of SAUSAGES

Although SAUSAGES is written in C++, SAUSAGES plans, links, and interface functions are specified using a C++ Lisp interpreter. The most notable variation of this Lisp is that it has a simple class system which can be especially useful in specifying types of "links" for the Link Manager. For example, there can be a "trajectory" link class that follows a trajectory and contains much of the administrative work for following trajectories. Then subclasses of the trajectory class can be specified that follow roads or go cross country. At present, we only allow inheritance of data and methods from a single parent class, but Figure 35 shows how even single inheritance can create a useful class hierarchy.

Another key feature of our interpreted Lisp is that it is trivial to write new C++ functions and to invoke them from Lisp expressions. This extensibility supports our philosophy of using Lisp as an interface and doing any computationally expensive work in C++. This takes advantage of the run-time strengths of both languages: Lisp's flexibility and C++'s efficiency.

Using Lisp as the interface to SAUSAGES allows great flexibility in specifying a plan or algorithm. Link entrance and exit actions are naturally specified by Lisp expressions. Arbitrarily complex Lisp expressions can be used in specifying "rule" criteria. Classes can be used as link templates, so a "follow-road" link is just an instance of the follow road link class with the appropriate data slots filled in and the remaining data slots filled with default values.

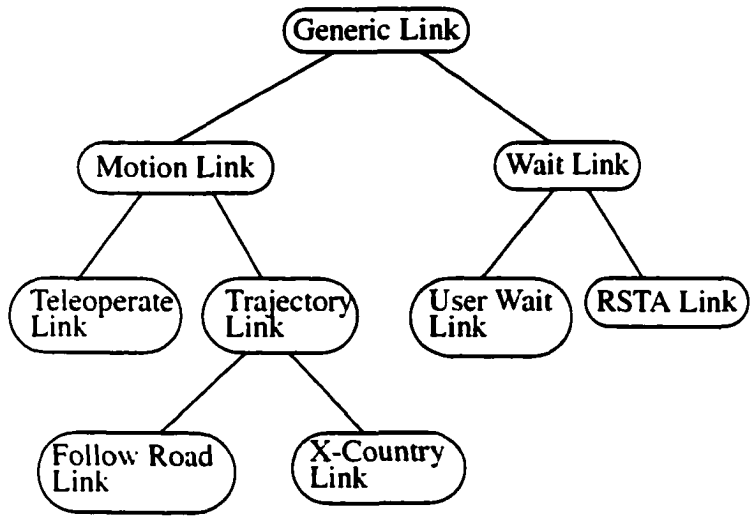


Figure 35 The UGV link class hierarchy

The most important module in SAUSAGES is the Link Manager, which actually manages the execution and monitoring of links. Link networks, i.e. plans, that the Link Manager will run can be read in from files or transmitted from other modules via a TCP/IP connection.

In order to run links and affect their behavior the Link Manager maintains a simplified production rule system. Each rule in the system has a list of tags, a criteria expression (the "left hand side" of the rule), and an action expression to be evaluated when the criteria expression evaluates to non-nil. A rule's criteria expression is only evaluated when one of its tags is marked as changed. This is not a complete forward propagating rule system, but it has proven sufficient for the purposes of mobile robot planning. If a complete production system is needed it will be easy to add and integrate. The Link Manager also maintains a global timer table which maintains a list of expressions to be evaluated at regular intervals

These global systems for productions and timers are the means by which a SAUSAGES plan can seem to be multi-threaded. When a link is run, the Link Manager adds the link's rules and timers to the global systems and when the link finishes they are removed. If two links are active "concurrently" both links' rules and timers are active in the global systems.

External modules can easily send the Link Manager expressions to be evaluated, and thus can easily affect the firing of rules in the global production system. The remote expression evaluation

provides one method for an external module to create and run links remotely. The Link Manager also provides methods for exporting references to the links themselves so that external modules can modify them as needed. The Link Manager can also send messages to external modules just as easily as it gets them. These are just some of the ways a SAUSAGES system can interact with external perception modules and plan monitors.

6.6 SAUSAGES example

I will present a simple example that will illustrate how SAUSAGES is used to quickly develop and test integrated systems through scripts and plans. The specific scenario will be a taken from the real military scenarios used in the UGV demos: An army scout has to follow a road until it gets to the front line, and then must use various cross country navigation techniques to drive to various positions to attempt to observer the enemy.

6.6.1 Action links

The mission will involve properly sequencing several different classes of actions. There will be road following, cross country navigation, teleoperation, and waiting for events. Each of these links exists in a class hierarchy given in Figure 35. The system integrator incrementally builds these links to ensure that they start the right modules in the right order with the right sequence to correctly implement the action. To understand how SAUSAGES implements these link classes we must understand the classes themselves as well as all of their superclasses. I have left out many details that are irrelevant to the example.

Some of these actions use the geometric query and trigger resources provided by the annotated maps system, described in Chapter 7.

Motion action

Defines the basic class for actions which move the vehicle.

Parameters: Initial speed of the vehicle.

Trajectory action

A trajectory action has a nominal, or expected, path of execution. It inherits all of the

aspects and parameters of the motion action class.

Parameters:

- Points: The points which make up the nominal trajectory
- Tolerance: How far from the nominal path the vehicle can stray before there is an error condition

Productions:

- If the vehicle has strayed from the trajectory by more than the tolerance, then fail.
- If the vehicle has completely traversed the trajectory, then finish.

Entrance actions: Initialize geometric monitoring for trajectory tolerance and trajectory completion.

Exit actions: Deactivate geometric queries.

Follow road action

Follow a road quickly using only road following given by a nominal trajectory. The link inherits all of the aspects and parameters of the trajectory action class.

Entrance actions:

- Start the road following module
- Initialize the weights with DAMN so that it listens primarily to the road follower

Exit actions: Stop the road following module.

Productions:

- Monitor for a failure in the road following module or a drop in the road following module's confidence in the road.

Cross country action

Go across country nominally following a trajectory. This tolerance on the trajectory will be very loose, since the system integrator should expect detours around unexpected obstacles.

Entrance actions:

- Start the local obstacle avoidance behavior
- Start the goal seeking behavior.

- Initialize the weights with DAMN so that it combines the output of the two.

Exit actions: Stop the obstacle avoider and the goal seeker.

Productions:

- Monitor for a failures in the local obstacle avoidance system such as being stuck or sensor problems
- Monitor for failure to progress towards the goal.

Teleoperation action

Teleoperate until an external module, usually a graphical user interface, sends a message which sets the variable **user-finished** to true. This class is not descended from the trajectory action, but rather straight from the motion action, because there should be no limitations on where the user can teleoperate the vehicle.

Entrance actions:

- Start the teleoperation behavior.
- Initialize the weights with DAMN so that it listens to the teleoperator.

Exit actions: Stop the teleoperation behavior.

Productions: If the event variable **user-finished** becomes true, finish.

Wait action

Pause the vehicle until something happens. This link is the basis for portions of the plan for which the vehicle is stationary.

Parameters: A message to print to the graphical user interface.

Entrance actions:

- Stop the vehicle
- Set DAMN weights to all 0
- Print the message

Wait for user action

Pause the vehicle until a graphical user interface causes the variable user-finished to go to true. Inherits other aspects from the wait action link.

Productions: If the variable **user-finished** is true, finish.

6.6.2 Plan links

By themselves, the action links can be sequenced into complex plans to perform interesting scenarios. Indeed, much of the work of the system integrator will be done by arranging simple sequences of these primitive actions to debug all of the details of link execution and transition.

But, we would like to build flexibility and the ability to recover from simple errors into the plan. SAUSAGES can satisfy this need through composite links, i.e., links which themselves consist of links or networks of links. I have developed a composite link which packages some simple contingencies and reactions to failure in a coherent, standard form.

The composite link, which I call the plan link class, consists of three sub-links:

- The action link, i.e. the link that embodies what the plan link should be doing, i.e., road following, or cross country traversal.
- The failure link, i.e., what action should be taken if the action link fails.
- The suspend link, i.e., what to do if the user suspends the plan temporarily.

Figure 36 gives the structure of the link network contained by the plan link. The plan link first starts the action link to perform the basic action. The plan link has a method which is configured to catch failures in the body links. So, if the action link fails, that failure is passed up the hierarchy to the plan link's failure method which sets the variable **action-failed** to true, and finishes the action link. The plan link also has a production monitoring the status of the **suspend-action** variable. If this variable goes to true, e.g. through an external graphical user interface sending a message to the link manager, then once again, the plan link finishes the action link.

The composite plan link defines the action link with a complex "next expression" field. This field is a conditional statement which determines what link to execute after the action link finishes. When the action link finishes, if **action-failed** is true, the next link is the failure link, and if **suspend-link** is true, then the next link is the suspend action. If neither is true, the whole compos-

ite link finishes and the plan goes on to the next plan link. Both the failure link and the suspend link return to the action link when they are completed

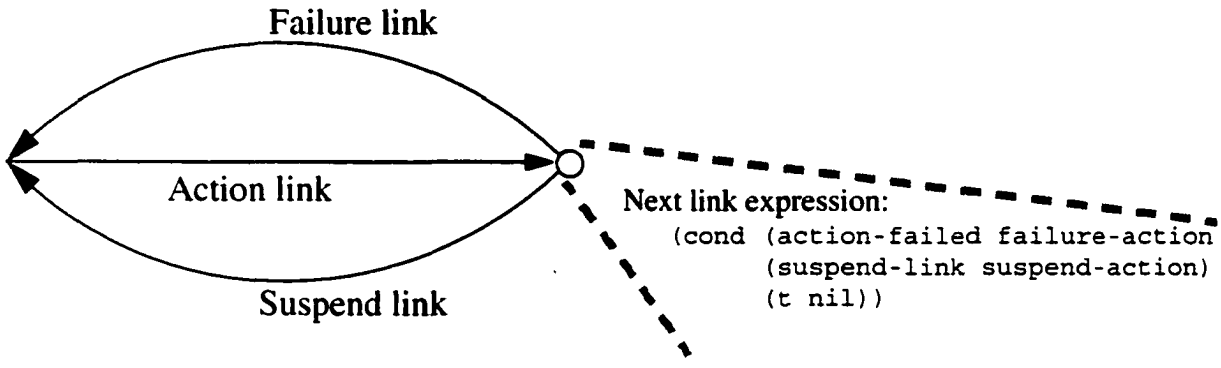


Figure 36 Composite plan link

Finally, the plan link monitors for changes in the variable **abort-link**. If this variable goes true, that means that the user gives up on this section of the plan, and whatever sub-link is being currently executed is completed, and the plan proceeds to the next plan link.

I define a hierarchy of composite plan links that corresponds to the hierarchy of the action links. For example, a road following plan link is simply a subclass of plan link with the action link fixed as a road following action. Although the failure links and suspend links could be anything, for every member of the hierarchy for this mission we define the failure link as a teleoperation action and the suspend link as a wait for user action.

6.6.3 The plan

Figure 37 shows a simple scenario that could be implemented using this set of links. The mission is to follow a road to a embarkation point, then move across country to an observation point. From that observation point the vehicle with wait, and perform RSTA (Reconnaissance, Surveillance, and Target Acquisition). A RSTA link is simple a wait for user link which starts up and monitors RSTA processes. The result of the reconnaissance is that either an enemy is observed, or

not. If an enemy is observed, then the plan exercises the contingency of going to one point. If not, then the plan directs the vehicle to go to a different point.

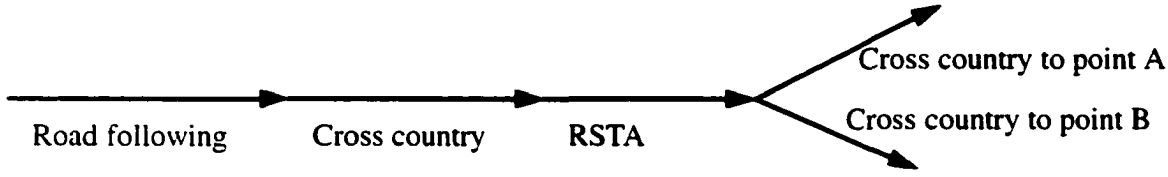


Figure 37 A sample SAUSAGES plan

6.7 SAUSAGES and planners

SAUSAGES is not a planning system. It is a plan executor and monitor that stands between a planner and the real world, between the world of symbols and propositions and the world of pixels and actuators. It depends on external sources for its plans, whether that be a human hand coding a mission plan for a specific demo or a sophisticated AI planner generating plans as it responds to changes in environment and goals.

One of the historical models of planning involves a separation of planning and execution [Nilsson, 1984]. A planner comes up with a complete plan, sends it to an executor which does its best to perform the plan, and reports success or failure to the planner. In this paradigm there is a conceptual wall between planning and execution, and many researchers have shown the limitations of this approach.

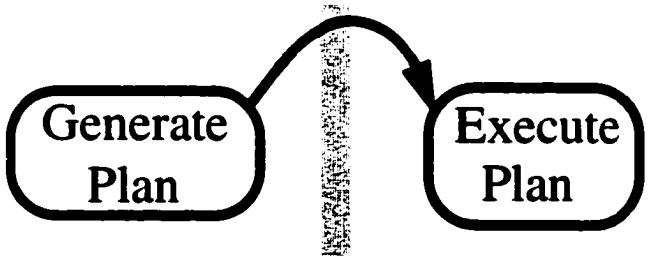


Figure 38 Separation of planning and execution

It is easy to construct SAUSAGES systems using this paradigm, but SAUSAGES is not limited to it. SAUSAGES was designed for heavy interaction with any number of external planners, not just one omniscient planner. External modules can create, edit, and run links. Links can be

designed to report status back to external modules and can have data associated with them that is not necessary for plan execution, but which do encode information necessary to manipulate the plan.

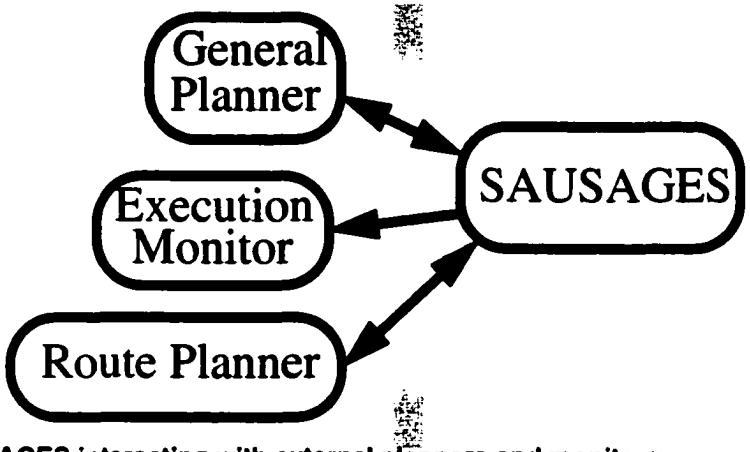


Figure 39 SAUSAGES interacting with external planners and monitors

A SAUSAGES plan does not have to be a passive, static graph, since the "next link" function can create new links at run time. The relevant case for planning is a next link function that generates a call to an external planner asking what to do now. The most extreme form of this approach would be to make every link's next function report status back to an external planner and request that the planner generate the next link.

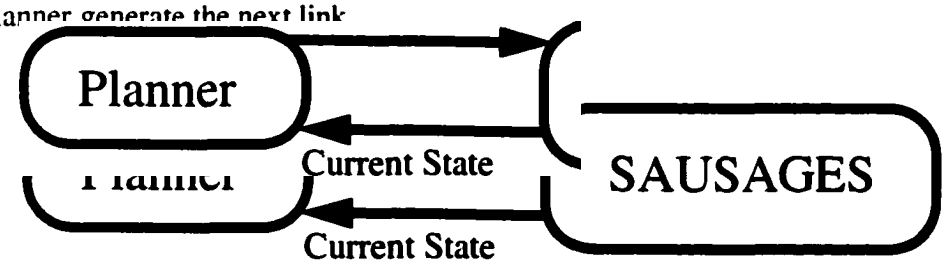


Figure 40 A SAUSAGES system that completely interleaves planning and execution

This approach, which totally interleaves planning and execution, can be used to interface a SAUSAGES system with incremental planners such as the Procedural Reasoning System (PRS) [Georgeff and Lansky, 1986] which may use the universal planner approach [Schoppers, 1987]. At each step in a plan an incremental planner examines everything it knows about the situation before applying the next action. This planning paradigm lets the system easily respond to errors, system failures, goal changes, and other unexpected changes in the world. It also lets the "plan" be mostly incomplete and underspecified. Using this planning/execution paradigm in a SAU-

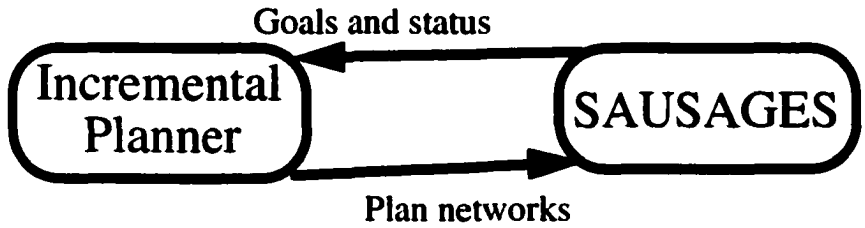


Figure 41 The ultimate SAUSAGES intelligent agent

SAGES system essentially reduces the role of SAUSAGES to a "mode manager," i.e., the planner spits out a high level command such as "follow the road using system X" which SAUSAGES translates into actual commands to perception and actuation modules. SAUSAGES is also used in such a system to translate failures of perception and actuation modules into symbology that the incremental planner can process.

While this approach is robust, it throws away many of the abilities of SAUSAGES in order to have everything controlled by a single monolithic planner. We find in building systems that putting too much of an emphasis on general purpose modules results in degraded real time performance. It is much better to have multiple specialized systems, with each system doing its job efficiently and robustly. SAUSAGES is not a general purpose module: its specialization is as a plan executor and monitor, and any computationally expensive job should be delegated to an external module. The "incremental plan" scenario does not take advantage of the things that SAUSAGES does well.

A better approach for the ultimate SAUSAGES system is to still use something very much like PRS, but instead of generating single plan steps the system would generate algorithms and plan sequences that are appropriate for achieving the goals of the mission. This plan graph sent to SAUSAGES would incorporate the necessary information to monitor and to detect failure. Any failures would be handled by reporting to the Planner the new state of the world so that it could generate a new algorithm or plan sequence to achieve the goals of the system with the new knowledge of the failure.

This approach takes advantage of the "reactive" abilities of an incremental planner like PRS while still exploiting the efficiency and flexibility of SAUSAGES. The question of which system is dominant, the incremental planner or the SAUSAGES plan executor, is irrelevant. It doesn't

really matter whether there is a SAUSAGES plan network that invokes the incremental planner which in turn generates SAUSAGES programs or if there is an incremental planner which generates a SAUSAGES program which calls out to the planner which in turn generates more SAUSAGES programs.

A human problem solver executing a task does not typically think about each step. A human will see a goal to achieve and will try to find a schema or algorithm that achieves that goal. Typically, a human problem solver will only realize that an algorithm is not appropriate when it fails, and only then will the slow, thoughtful step by step "planning" to achieve the goal or work around it be tried. This behavior, known as Einstellung effect [Luchins, 1942], corresponds well to a system that uses an incremental planner to choose SAUSAGES plan networks. The incremental planner has a goal, and to achieve that goal it generates a SAUSAGES network. The SAUSAGES network runs until it finishes or fails. If the network successfully finishes, the planner can go on to working on the next goal. If the network fails, the planner can then go into a more intensive step by step incremental planning mode, or it can take whatever other course of action it deems necessary.

6.8 Conclusions

The purpose of SAUSAGES is to provide system integrators with a tool to quickly prototype, test, and modify demonstrations of a systems capabilities. With SAUSAGES a system integrator can quickly create a flexible script for a scenario. SAUSAGES lets the integrator easily test all of the individual actions and transitions and tune them to perfection so that when all the pieces are put together the system performs a useful task.

SAUSAGES plans can be arbitrarily complex and flexible, but there will come a point when SAUSAGES alone is not the right representation. For example, if what the system really needs to do is reason about goals and methods, then SAUSAGES alone is not appropriate. SAUSAGES is appropriate when there is a set of known methods for goals, and all that needs to be done is link those methods together. SAUSAGES can easily handle simple contingencies or looping until a certain condition is met. SAUSAGES is not appropriate when the plan is a list or network of goals, and for each goal there must be reasoning done to match the right method to the goal. SAU-

SAGES links can be made to mimic this functionality, but the result will never be as elegant as a real reasoning system such as PRS.

One point of emergent architectures is that when a part is no longer appropriate, it can be changed or replaced easily to adapt to the new requirements. Since most of the modules in the system are isolated from the particular architecture, SAUSAGES could be replaced with a complete reasoning system without causing a large amount of work for anyone except the developer of the reasoning system. The change would only require new reconfigurable interfaces to the new reasoning system.

On the other hand, the emergent architecture approach gives a compelling reason for keeping SAUSAGES in the system even once the need for a more sophisticated reasoning system arises. SAUSAGES provides a mechanism for isolating the researchers building reasoning systems from the actual nitty gritty details of the system. This separation is extremely useful when the reasoning researchers are not the system integrators.

In most systems the developers of the reasoning systems have been the system integrators, so the separation has not been an issue. The confounding of system integration and planning research comes from two sources.

- History: Developing and proving reasoning systems has been the motivator for many robot systems, and thus the developers of the reasoning system become the system integrators. This is the opposite time line from my approach, which is given a task, find plan executors or reasoning systems that are appropriate.
- Convenience: For a reasoning system to work it needs to know how to interact with every module in the system. A reasoner is useless if it does not know how to get status and events from the other modules in the system and how to effect action on the system. Thus, in the worst cases the researchers building reasoning systems have to know just as much about all of the details of how the system works as the system integrators. In these cases it is convenient to have the reasoning researchers be the system integrators.

SAUSAGES provides a middle ground between planning and action. It lets system integrators build a set of links that have been fine tuned and tested to actually work on real vehicles. The sys-

tem integrators can hook these links together into complex, if somewhat rigid, plans, and all of the details of transitioning between the links can be ironed out. The system integrator can work on what the real error conditions for the system are and how to detect them, if not how to handle them elegantly. The result can be considered a "language" that a high level reasoner can use and extend to make the system more robust and autonomous.

SAUSAGES allows the development of a complete outdoor mobile robotic system that is driven by what the robot can perceive rather than what the planners want to perceive. As the perception and actuation modules improve, the developers create SAUSAGES links and algorithms to perform various tasks. When a planning system matures to the point of being integrated with a real robot it can use these various SAUSAGES links and algorithms that have been heavily tested in the field as building blocks for plans.

SAUSAGES does not lock the system designer into any single planning or architectural paradigm. It is simply a bridge between planning and action, and does not specify how either is to be accomplished, or if either is to happen at all. SAUSAGES lets the system designer vary the level and extent of involvement of a general planner in the execution of a plan.

For example, suppose that one leg of a mission has a pair of vehicles performing a "bounding overwatch." A bounding overwatch is a maneuver in which two vehicles mutually support each other as they travel under the possible observation of an "enemy." A bounding overwatch cannot be completely specified as a set of trajectories and mode transition points, since the exact execution depends on a large number of environment variables and constraints that can not be determined a priori. A SAUSAGES system could implement a bounding overwatch by having a general purpose reactive planner keep track of the current goals and states of the vehicles and enemy. This planner would generate new links one by one as the bounding overwatch progresses. Alternatively, if a system designer can design a bounding overwatch algorithm it would be more efficient to have a specialized bounding overwatch module implement this algorithm. In this case, the higher level planner just generates a link, "do a bounding overwatch," which SAUSAGES interprets as "ask the bounding overwatch planner to do a bounding overwatch." If the algorithm for doing a bounding overwatch is simple enough, the bounding overwatch module could be completely dispensed with, and the bounding overwatch could be done through an algorithm imple-

mented as a SAUSAGES network. With SAUSAGES, any of these could easily be implemented, and it is up to the system designer to decide which is best.

SAUSAGES provides a module that can be the middle ground between the extremes of the robotics world. It allows a researcher working on a high level planner to avoid the awful implementation details of a real mobile robot system by abstracting the operation of the lower level perception and actuation modules. It allows a system integrator to easily build a system that showcases the perception work in a complex scenario without worrying too much about issues of planning and error recovery that are irrelevant to the task at hand. Finally, SAUSAGES facilitates the joining of these two extremes into a complete, intelligent, and robust outdoor mobile robot system.

Chapter 7 Maps and Plans

7.1 Better plans through geometry

An intuitive way to represent a plan and execute it is event driven, i.e., "Follow this road until the second intersection, turn left, and stop at a mail box." This plan representation works well for simulated and indoor mobile robots, which operate in a highly structured and controlled environment where events necessary to the plan execution are easily perceived. Unfortunately, once a robot system leaves the laboratory it is in a much less structured, less controlled, and less predictable environment. The events that an outdoor mobile robot system needs to detect to execute a plan are very difficult to discriminate from the background environment. For example, only recently, years after starting work in outdoor mobile robots at Carnegie Mellon, have we developed a reliable intersection detector. Just the development of the perception modules for motion, such as road followers and obstacle avoiders, takes years of effort. If we had waited for all the perception capabilities necessary to execute an event based plan to be developed, we would still be waiting, and we would still not have a system capable of performing complex, multi-modal missions.

In order to sidestep the problem of building the event detectors necessary to perform complex missions, we take advantage of the fact that our vehicles have relatively reliable positioning systems, and especially have accurate odometry sensors. Instead of relying on undependable or nonexistent event detectors to sequence a mission, we use an annotated map, which converts the

problem of looking for a particular event to one of looking for a particular position of the vehicle [Thorpe and Gowdy, 1990, Thorpe et al., 1991].

Annotated maps are another system engineering tool to craft architectures that perform real tasks. Annotated maps make the problem of building systems that perform complex, interesting tasks tractable even though all of the support framework for an event based mission such as object recognizers and situation assessors may not be in place.

7.2 Annotated maps

Much of the information that mobile robots need is tied directly to particular objects or locations. Maps, object models, and other data structures store useful information, but do not organize it in efficient and useful ways. An annotated map is a map-based knowledge representation which indexes information to the relevant object and locations. The annotations are used for a wide variety of purposes: describing objects, providing hints for perception or control, or specifying particular actions to be taken. Annotated maps also provide query mechanisms to retrieve annotations based on their map locations, as well as "triggers", which cause a specified message to be delivered to a particular process when the vehicle reaches a given location in the map.

These annotated maps serve a crucial role in enabling missions that are otherwise beyond the reach of our systems. Control descriptors allow mission planners to specify what the vehicle is to do at particular locations, reducing the need for onboard planning. Object descriptors contain detailed instructions of how to recognize a particular object, or contain the appearance of this object as seen by a particular sensor on a previous vehicle run. Such information greatly simplifies the problem of seeing and recognizing objects. Geometric queries enable the vehicle to focus its attention on objects in its vicinity, reducing database access and matching time. The trigger mechanism frees individual modules from having to track vehicle position, allowing them to devote their processing to the task at hand or to lie dormant until they receive their trigger message.

Annotated maps do not by themselves solve difficult problems of sensing, thinking, or control for autonomous vehicles. Their contribution is to provide a framework that makes it easy for other

modules to cooperate in planning and executing a mission. Annotated maps thus fill a need that is common to many different vehicles, missions, and architectures.

Many analogous annotated maps exist for human use. Aeronautical navigation charts contain symbolic descriptions of routes (airways) and landmarks, and include annotations such as the Morse code call letters of radio navigation beacons. The AAA produces "Triptiks", which include annotations for route, current conditions ("construction", "speed check"), road type (interstate, two lane, etc.), general conditions ("winds through rolling hills"), points of interest (rest areas, gas, food, and lodging) etc. An intelligent person can usually drive a route without such aids; but they do provide a convenient framework for preplanning, and make "mission execution" easier. Furthermore, as we drive a route, we build our own mental representations of landmark appearance, curves in the road, and so forth, which we use to follow the same route more easily at a later time. Our annotated maps provide the same kind of functionality for autonomous mobile vehicles.

7.3 Maps and robotics

Maps have been a fundamental research area in mobile robotics. Maps can encode what a robot knows about its environment, and are often used to model the world and make decisions about how to achieve goals. The two most common approaches to world mapping in mobile robots are *grid-based* paradigms and *topological* paradigms.

In grid based approaches, the world is represented as cells[Elfes, 1987, Borenstein and Koren, 1991]. Each cell represents a discrete part of the robots environment and holds information about that cell, such as whether that cell contains obstacles or not. Any drift or systematic inaccuracy in the robot position is disastrous to a grid-based world map, so the robots position must be precisely tracked as it moves through its environment, either through introspective sensors such as geometry or inertial navigation systems, or through matching sensed landmarks to known landmarks[Thrun, 1998]. In addition, the amount of memory consumed by a grid-based maps quickly grows with the size of the mapped environment, since the resolution of a grid must be fine enough to capture every salient detail of the world.

In a topological map, the world is represented as a graph[Kuipers and Levitt, 1988, Mataric, 1990, Kortenkamp and Weymouth, 1994]. Nodes in the graph correspond to distinct locations, situations, or landmarks, and two nodes are connected if there exists a direct path between them. The position of the robot in a topological model is not dependent on a global frame of reference as with grid-based approaches, but rather is relative to the landmarks specified in the nodes. Topological maps are typically much more compact than grid-based maps, since they only represent the "important" aspects of the world. Unfortunately, topological maps can be difficult to construct, maintain, and use in a large-scale environment in the face of ambiguous sensor information. If landmarks are misrecognized, then the robots estimate of its position in the topological model can become inaccurate, and thus the actions taken by the robot can become invalid.

There is a key difference between the usage of either of these map paradigms and annotated maps: the typical map, either grid-based or topological, is used to represent the world in order to come up with a plan of action; an annotated map *is* a plan of action. An annotated map is not used as a resource for planning, but its annotations implicitly define the plan.

Annotated maps are essentially grid-based maps, but the cells can be fairly coarse, on the order of one or two meters across. Each cell does not contain just traversability information, as with an occupancy grid[Moravec, 1988], but rather a list of arbitrary annotations concerning that cell. The information can be complex, such as describing how best to image a landmark expected in the cell, or the information can be simple, such as a trigger to cause a message to be sent to a specific module when the robot enters the cell. The annotations in the coarse global map provide advice and information to modules, such as obstacle avoiders, landmark detectors, and road followers which maintain a much finer local representation of the world.

An annotated map represents a path through the world, it is not meant to represent the world as a whole. Thus, if the path is globally inaccurate, for example if the mapped end-point of the plan is dozens of meters from the actual plan, but the path is locally accurate, for example at any given point along the path the landmark locations relative to the vehicle are no more than half a meter off, then using the annotated map will bring the system from its start-point to its end-point. The annotated map will be almost completely useless as a planning resource, that is it could not be used to plan an alternate route from the start-point to the (globally inaccurate) end-point.

As in topological maps, an annotated map system is constantly updating its position relative to its world representation using sensor input such as road detectors and 3D object mapping, thus allowing the system to overcome systematic position errors. Unlike most topological maps, an annotated maps system also combines the metrics of the map with the metric information of the positioning system. Our annotated maps system acknowledges that, for all its faults, the most stable sensing modality for our outdoor mobile robot is its position sensing. Only by deeply integrating position sensing with landmark detection, which is fraught with errors and ambiguities, can we build a successful system.

7.4 Implementation

In general, planning and executing a mission requires several types of knowledge: what to look for, and how to see it; what to do, and how to accomplish it; where to go, and how to get there. The knowledge may range from high-level symbols, to low-level raw data. Knowledge is both internal to a single module, and used by controlling modules to switch between knowledge sources. Approaching the intersection, for instance, the perceptual knowledge includes:

- symbolic: intersection
- geometric: size and shapes of intersecting roads
- sensor-specific: use laser range finder to pinpoint the position by landmark identification
- raw data: landmark 2 meters tall, 0.4 meters wide at position (x,y)

Control knowledge can also span a range of levels:

- symbolic: turn left at intersection
- geometric: intersection angle 45 degrees
- vehicle-specific: turn with a circular arc of radius 15m
- raw data: steering wheel position left 1200 clicks

This knowledge must be carefully organized if it is to be useful. If the vehicle has to sort through all bits of information it has about every possible object, it will overshoot the intersection

long before it has figured out how to recognize it or deduced that it was supposed to turn. It is far better to have information tied directly to the map, or automatically retrieved as needed. The landmark recognition module, for instance, must be able to ask for a description of objects within its field of view, and retrieve the knowledge it needs to recognize them.

7.4.1 Map resources

The annotated map system consists of a database library, which maintains a local "list" of objects in the map, and a set of resources. These resources are servers that clients can talk with via the communications library. The two most significant resources are the Annotated Maps module, which is used as both a means to efficiently query the object database and as a positional alarm clock, and the Navigator module, which is used to maintain the best estimate of the vehicle's position on the map.

The object database contains all the information about the world. The map can contain the following types of objects:

- Generic objects: Discrete circular or polygonal objects, e.g., trees, mailboxes.
- Road points: Road points are linked together to form roads
- Intersections: Intersections link different roads together
- Meta objects: Meta objects are used primarily for global information about roads. For example, a road's meta object might contain the name of the street.
- Alarms: These are artificial objects placed in the map for control purposes.

Each object in the database can be annotated with extra information that individual clients in the database have to interpret.

The Annotated Maps module takes the list of object descriptions and forms a two dimensional grid so it can answer queries about the object database more efficiently. For example, the object matcher has to know what objects are in its field of view. To find this out it could look through the object database, checking for objects that overlap the field of view polygon, but this would be very inefficient for a large database. Instead, the recognizer can send the polygon to the Annotated Maps module which can use the grid to quickly find the objects within the polygon.

The Annotated Maps module is not just a passive geometric database, it is actively involved in the control of the system. The alarm objects are placed in the map by a human user to plan the scenario. Alarms are conceptual objects in the map, and can be lines, circles, or regions. Each alarm is annotated with a list of client modules to notify and the information to send to each when the alarm is triggered. When the Annotated Map manager notices that the vehicle is crossing an alarm on the map, it sends the information to the pertinent modules. The map manager does not interpret the alarm annotations: that is up to the client modules.

Alarms can be thought of as positionally based production rules. Instead of using perception based production rules such as, "If A is observed, then perform action B", an Annotated Map based system has rules of the form, "If location A is reached, then perform action B". Thus we reduce the problem of making high level decisions from the difficult task of perceiving and reacting to external events to the relatively simple task of monitoring and updating the vehicle's position.

The first step in building an Annotated Map is collecting geometric information about the environment. We build our maps by driving the vehicle over roads and linking the road segments together at intersections. Once a road is built, the human user can create a "meta object" for that road which describes the whole road. In this system, we annotate a road's meta object with a short description or label and with the absolute orientation of the vehicle at the start of the road. At the same time, a laser range finder is used to record the positions of landmarks such as mailboxes, trees, and telephone poles. The landmark collector also annotates each object it finds with the two dimensional covariance of the object's position, in addition to its location and description, which can be used later by the object matcher. After the map building phase, a human planner draws "trigger lines" across the road at various locations. For example, the person knows that when approaching an intersection, the vehicle should slow down, and so chooses to put the trigger line at the approach to an intersection. The trigger line goes across the road at that point, and is annotated with a string of bits that represents the new speed of the vehicle. During the run, when the vehicle crosses the trigger line, the map manager sends the string of bits to a module that interprets the information and slows the vehicle to the desired speed. Depending on its content, an alarm could also be interpreted as a wake up call, as a command, or even as simply advice.

7.4.2 Position maintenance

In our implementation, the Navigator maintains a correction transform rather than the actual position. The vehicle controller provides positioning information completely independently of the map. This positioning information is derived from gyros and dead reckoning, and is in an arbitrary frame of reference that we call the controller coordinates. Client modules that do not need the map can use positioning information in controller coordinates. For example, a road following vision system might be interested only in the motion of the vehicle between frames, since it does not need to know the absolute position of the vehicle on the map, so it can directly use the controller position to get relative motion. Other modules, which need to use the map, need the transform T_{world} which converts controller coordinates to map coordinates. This T_{world} transform is maintained by the Navigator. Thus, a module which needs to know the current position of the vehicle on the map must first acquire the current T_{world}, then query the controller for the current vehicle position in controller coordinates, then apply T_{world} to convert to map coordinates.

There are several advantages of this system. First, the controller coordinates are never corrected, so modules do not have to worry about jumps in relative locations of nearby objects that are stored in controller coordinates. In other schemes, when a position correction happens all objects stored in the current local coordinate system must be updated. Second, the position corrections which do occur (when the T_{world} is updated) can occur in a separate, possibly slower process (the Navigator), without interrupting the real-time loops of the controller. This decomposes our system logically into a fast system running in local coordinates, and a separate set of often slower processes that need to refer to the map. Besides being a good design tool, this is also a practical help in doing the updates. Time is no longer as important a quantity. Directly updating the controller's position would require changing the position estimate in real time, while the controller is still doing dead reckoning. In our scheme, instead, the Navigator's T_{world} can be updated with no real-time concern, since it is not rapidly changing. Also, filtering the relatively stable quantity T_{world} is relatively easy since it will typically require only small changes. This also means that the compounding operation, which increases positional uncertainty during dead reckoning between landmark fixes, is also simplified. The mean of the T_{world} stays constant during compounding, and only the positional covariance C_p must be changed.

7.4.3 Kalman filtering

There were two related problems with the simple position maintenance scheme. First, the corrections are made independently by each module, so the position estimate is completely determined by whichever module has made the most recent correction. Thus, if one of the corrections is slightly incorrect, it influences the entire system. Second, the position estimates carries no measure of error bounds. The landmark recognition subsystem, in particular, needs predictions from navigation to focus the attention of the sensing and the processing. Without a measure of the error bounds for a particular position report, the landmark recognizer does not know how far in advance to begin looking, or how large an area to search.

The solution is to use filtered updates, rather than absolute updates. A very simple Kalman filter gives approximate bounds on vehicle position, and could be used to integrate new measurements with current estimates, instead of completely throwing out the old position.

To estimate the vehicle's position as accurately as possible, odometry data, gyroscope heading data, laser range images, and road information are used in conjunction with previously recorded maps to estimate vehicle position and uncertainty. The vehicle heading is measured by a commercial gyroscope which employs its own Kalman filters to incorporate compass and gyroscope data to produce exceptionally accurate heading data. For this reason, our filter formulation is only concerned with the vehicle's x and y coordinates assuming perfect heading estimation from the commercial gyroscope.

The Navigator uses the real time controller's position estimation derived from the odometry and heading data as a basis for an internal transform developed from ERIM and map information. The Kalman filter is a black box to the Navigator which internally maintains the position uncer-

tainty. The Navigator invokes two operations in this formulation, the compounding and the merging. Figure 42 shows the covariance compounding and merging equations.

Compounding Equations		Kalman Filter Equations		
$C_P = J\begin{bmatrix} C_P & 0 \\ 0 & C_V \end{bmatrix} J^T$		$C_P = \left(C_P^{-1} + C_M^{-1}\right)^{-1}$ $\bar{P}' = \bar{P} + K(\bar{P}_M - \bar{P})$		
$C_P = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix}$	$C_D = \begin{bmatrix} \sigma_l^2 & 0 \\ 0 & \sigma_d^2 \end{bmatrix}$		$\bar{P}_M = \begin{bmatrix} x_E \\ y_E \end{bmatrix}$	$\bar{P}_M = \begin{bmatrix} x_R \\ y_R \end{bmatrix}$
$J = \begin{bmatrix} 1 & 0 & \cos(\theta) & -\sin(\theta) \\ 0 & 1 & \sin(\theta) & \cos(\theta) \end{bmatrix}$		$K = C_P(C_P + C_M)^{-1}$		

Figure 42 Kalman filter equations

The compounding operation is invoked at regular distance intervals of 0.5 meters. The covariance matrix for dead reckoning, C_v, contains the variance of lateral error σ_l and distance traveled σ_d. These variances are assumed independent. σ_l and σ_d are updated using the variances previously determined from odometry data and road follower performance. The new position covariance matrix C_p' is computed by using the Jacobian J to compound C_v and the old C_p.

The merging operation is invoked when another module provides an estimate of the vehicle position. In our current system, these additional estimates occur when the object matcher finds a new vehicle position from landmarks or when the road follower provides a steering command. The Navigator is given a vehicle position P_m measured by one of the perception systems and corresponding covariance matrix C_m. P_m and C_m are merged with the current estimates of position P and uncertainty C_p by the Kalman filter. This merging operation updates the covariance, C_p', and returns the updated position P'. The Navigator uses the new map position P' and the controller position (x_c, y_c) to generate a new T_{world}.

The landmark matcher directly gives the Navigator a new estimated vehicle position P_m and its covariance C_m. The Kalman filter module directly uses these values, plus the old position estimate P, as the inputs to its filter.

To update P_m and C_m from the map from road information, the Navigator follows a similar procedure. First, the Navigator uses the current transform T_{world} to estimate vehicle position P. It then projects P onto the road, to find the measured vehicle position P_m, since the road follower is assumed to keep the vehicle on the road center. Road updates can move the vehicle position perpendicularly to the road but not along the road. The Navigator therefore forms a covariance matrix C_m oriented along the road, with the road follower's error estimate perpendicular to the road and an essentially infinite variance along the road. As in landmark matching, the P_m estimate, the C_m covariance, and the P prior position estimate are passed to the Kalman filter module to generate a new P' and C_p. The new P' is used by the Navigator to build its new T_{world}.

7.5 Results

7.5.1 Extended run

A typical mission for our Navlab mobile robot is a delivery task on unlined, unmodified suburban streets. The Navlab has specialized perception modules, including color vision for road following on major roads[Kluge and Thorpe, 1990], dirt roads[Crisman and Thorpe, 1990], and suburban streets[Pomerleau, 1989]. It also has 3-D perception, using a scanning laser range finder, for landmark recognition and obstacle detection[Hebert et al., 1990]. Inertial navigation on the Navlab is accurate enough to drive blind for short distances[Amidi, 1990].

In order to accomplish its mission, the Navlab must use several of these modules. Road following using color vision will follow streets, but will not be able to recognize intersections. Inertial navigation will drive through intersections, but must have an accurate starting position. Landmark recognition will update vehicle position before intersections, but is too slow to be run continuously. Only a combination of all those modules, each running at the appropriate locations, will produce an accurate and efficient mission.

Figure 43 shows a typical annotated map of a suburban area, including about 0.7 km. of road with two T intersections, and a variety of 3-D objects. Object information was collected using the ERIM laser range finder, and the road information was collected by using the inertial navigation system to provide accurate vehicle positions while we traversed the route.

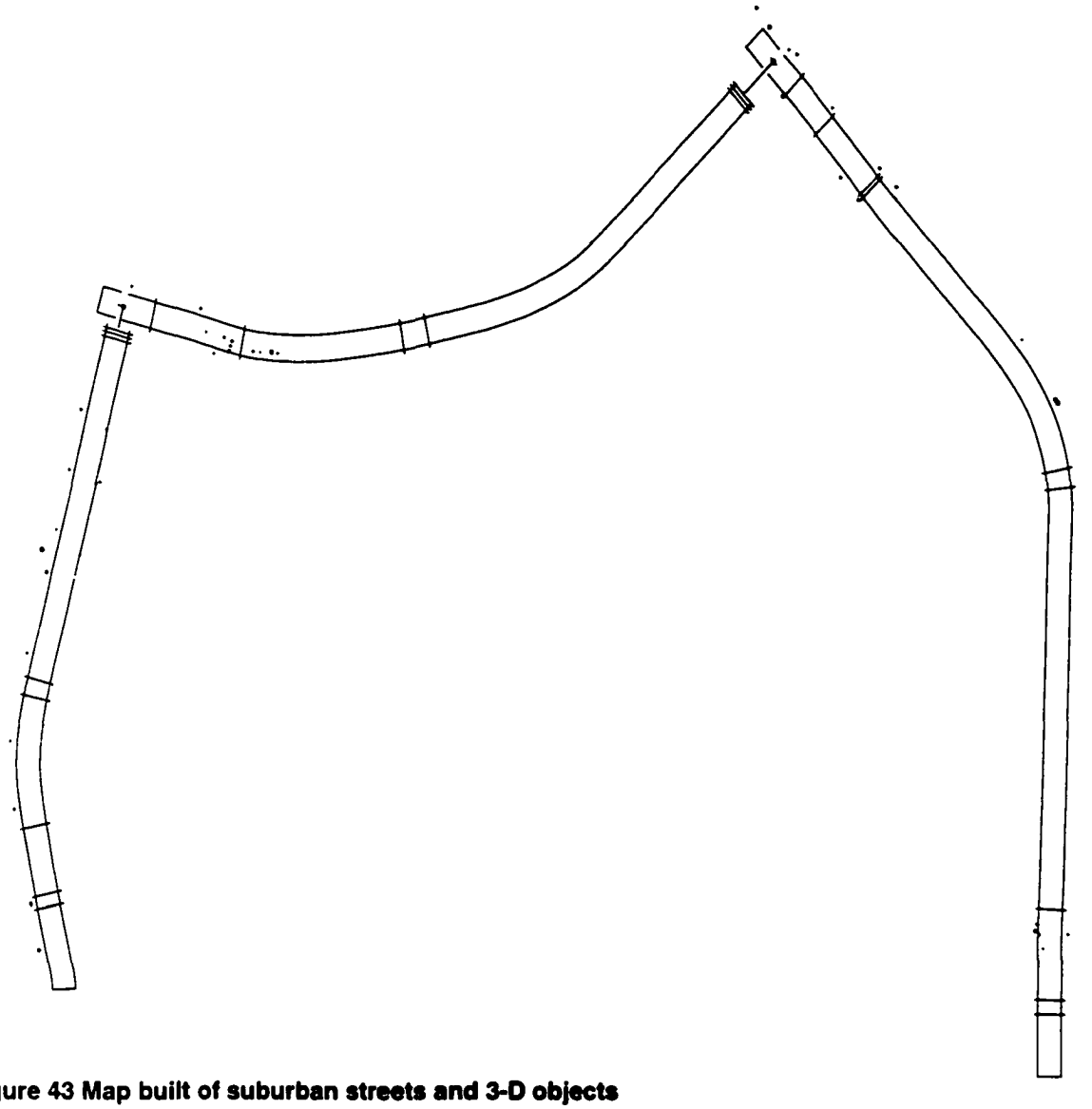


Figure 43 Map built of suburban streets and 3-D objects

The goal of this run was to drive from a house near the beginning of the map to a specified house near the end. Annotations were added to the map to enable the Navlab to carry out this mission. There were annotations to set the speed appropriately: up to 3.0 m/s in straightaways and down to 0.5 m/s in intersections. Other annotations activated and deactivated the module that uses the laser range finder to correct vehicle position based on detected landmarks. Before every intersection there was an annotation that switched driving control from a neural network vision program to a module that used knowledge from the map of the intersection structure and dead reckoning to traverse the intersection. Finally, there was an annotation at the end of the route that

caused the vehicle to stop at the appropriate object. The route was successfully traversed autonomously.

7.5.2 Kalman filtering.

We tested the system on a twisting park trail near campus. A human driver drove up the road to map it while the object detector found landmarks such as large trees. The object detector annotated each landmark with its observed covariance in x and y. After we built the map, we put one trigger line in it which tells the landmark matcher to start looking for landmarks during a run. One such map is shown in Figure 44.

To start a run, the driver drives the vehicle onto the road, and the operator then estimates the vehicle's initial position. The initial orientation is calculated by taking the difference between the angle currently reported by the gyros and the gyro angle annotated in the nearest road and adding that difference to the map angle annotated in the nearest road. The amount of drift in the gyros was small enough that this method of orienting the vehicle was much more accurate than just

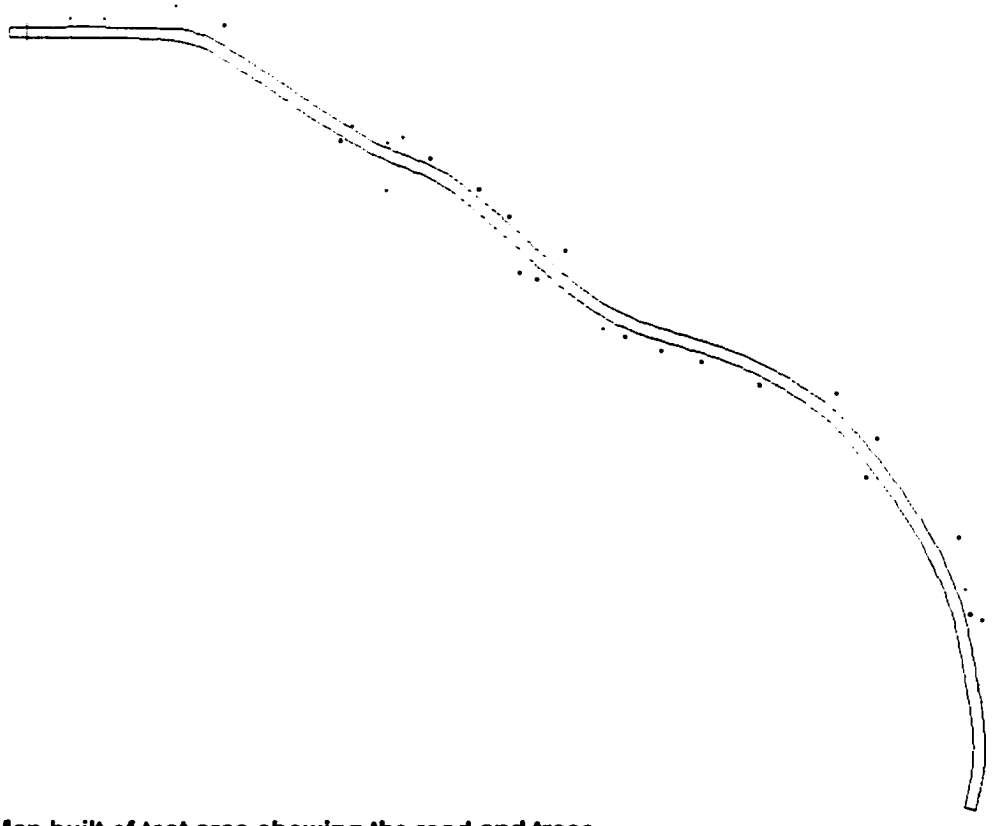


Figure 44 Map built of test area showing the road and trees

Figure 45 Vehicle position and uncertainty during a run

guessing the vehicle's orientation. We estimated the initial uncertainty of the vehicle's position as three meters in x and y. Figure 45 shows the uncertainty ellipse of the vehicle in the middle of the run after being shortened along the direction of the road by landmark matching and after being thinned by updates from the road follower.

One interesting result is that making lateral road corrections on a curving road can correct an error in position along the road. This is not immediately obvious intuitively since an individual road correction gives no information about the position of the vehicle along the road, only information about the lateral position of the vehicle on the road. We first noticed this effect by observing the changes in uncertainty ellipses during a run. In Figure 46, a vehicle has a circular uncertainty at position a, and after a road update at position b the uncertainty ellipse becomes a needle in the direction of the road. Imagine that the vehicle travels through the corner. At position c it has an uncertainty ellipse very similar to at position b, but after another road update at position d, the ellipse shrinks drastically. Simply from local road updates, we are vastly more confident about the position of the vehicle along the road at position d than at position a. This implies that

the apparently difficult problem of using road curvatures as landmarks is solved by simply making local corrections to the vehicle's position perpendicular to the road.

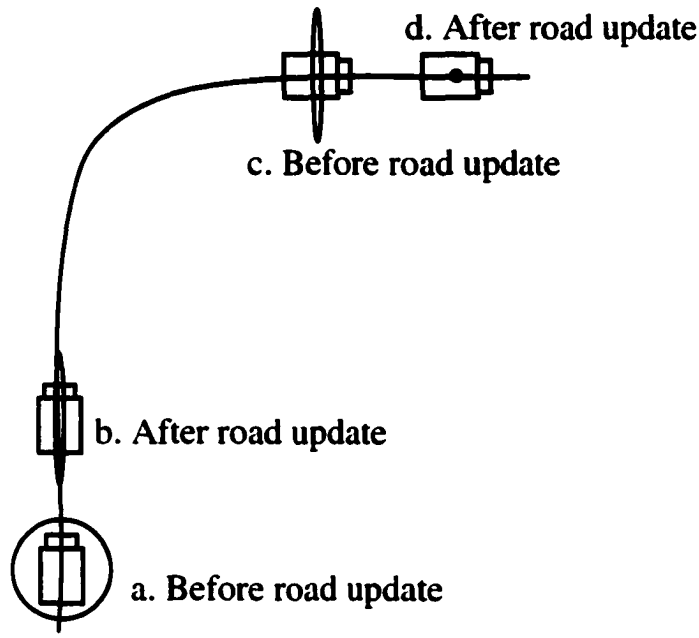


Figure 46 Effects of road updates on position uncertainty

This effect was evident in our runs. We deliberately started the vehicle in the wrong place in the map, displaced along the road by 4 meters. The landmark matcher could not match landmarks with this large an error, so it failed initially. After going through a curve with just road updates, the position in the map was close enough to the position in the real world that the landmark matcher was able to match sensed objects with map objects and further refine the position.

Are there cases in which the road updates confuse the position along the road? Unfortunately, yes. The autonomous road follower drives slightly differently than the human driver who made the map. For instance, the road follower may cut a corner slightly more than the human driver. Our model assumes that the vehicle is in the center of the road, even though it is not. A series of very slightly erroneous corrections going through a corner will result in an error in the distance along the map. Figure 47 illustrates the problem. This problem could be addressed by having the road follower be the driver when making the maps, or by increasing the compounding error in the

direction of travel when going through a curve, or by simply not performing road updates when the vehicle is going through a curve.

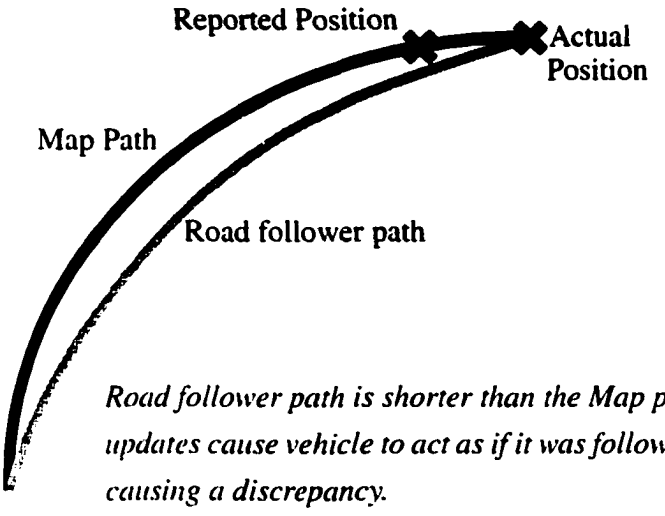


Figure 47 Errors caused by road updates during inaccurate driving through a turn.

We made a series of runs with the vehicle and measured performance qualitatively, and derived some preliminary quantitative measurements from system performance. Qualitatively, the most important criterion is that the landmark matcher should be able to find landmarks within its expected error. We found that to be the true in nearly all cases. Quantitatively, at the end of typical runs of 350 meters, the position updates generated by the matcher were usually between 5 to 10 cm.

7.6 Annotated maps and SAUSAGES

A system that only uses Annotated Maps for mission planning will rigidly perform one pre-planned mission, but won't react well to unexpected changes in the plan, goals, or environment.

For example, an Annotated Maps system can handle repeated, well established scenarios, such as mail delivery or a daily commute, it cannot begin to execute a simple, non-geometric plan for the first time, such as, "go to the first corner and turn left." For every mission there has to be an a priori, detailed map and a priori expert knowledge of how to navigate around in that map.

Another major limitation of an Annotated Maps "plan" is that it is only implicit, i.e., the sequencing of the "steps" of the plan is implicit to the geometric placement of the annotations. The Annotated Map manager itself has no concept of "plan", just of the map and the vehicle position. This makes it difficult to do things like loops in the plan without some serious and inelegant extensions to the map manager. It also makes it hard to monitor for failures of the plan. How is the map manager supposed to detect a failure in the plan when it doesn't even explicitly know what the plan is?

To solve these problems we integrated annotated maps with SAUSAGES. It is obvious in SAUSAGES how to represent event based plans: Each link terminates on the occurrence of some event, such as "intersection detected," or "Fred's mailbox found." Unfortunately, this brings up the original problem with this type of plan representation for outdoor mobile robots: Detecting the arbitrary events necessary to proceed from plan step to plan step. We can easily solve this problem by using segments of Annotated Maps as individual links.

In the simplest possible annotated map link, the only production rule would be, "If vehicle X crosses the line (x1, y1), (x2, y2), end the link and go to the next one." The next link would just be the next segment of the Annotated Map. Stringing links of this type together exactly simulates the capabilities of our current Annotated Map system, with the plan made more explicit.

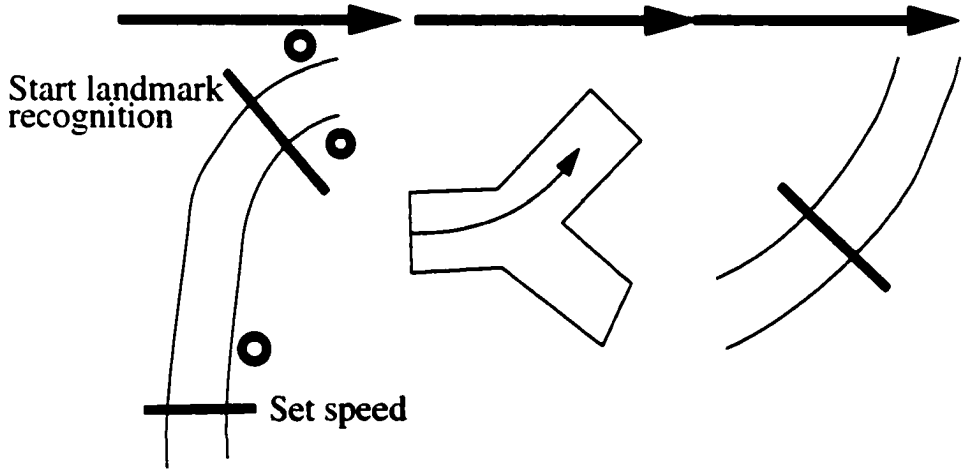


Figure 48 Using Annotated Maps as SAUSAGES links

Using SAUSAGES to make an Annotated Maps plan explicit frees a system from many of the limitations of Annotated Maps while retaining the central advantage of Annotated Maps, which is

the reduction of the event detection problem to a position monitoring problem. For example, retrotraversing a trajectory was a problem in Annotated Maps since the system had to discriminate between the annotations relevant to the journey out and the annotations relevant to the journey back. SAUSAGES easily solves this problem by allowing two annotated map links with the same geometric data, such as roads and landmark information, but with different annotation data for each leg of the traversal.

Since each step of the plan is an explicit link, the user can attach explicit and different failure conditions to each step. Thus we can easily and flexibly specify the conditions under which a plan fails and what to do when the plan fails. For example, in a road following annotated map link we would want a failure condition to be "if the vehicle position is too different from the road position, fail." For a cross country annotated map link we would want the same condition, but with a higher tolerance. Doing this is trivial in a SAUSAGES plan because each class of link can have a different set of production rules that indicate errors. With Annotated Maps alone even this simple error monitoring required adding code that didn't fit the design well.

To SAUSAGES, a link is a link, so it is trivial to mix "geometric," or map based links with "cognitive," or event based links. A SAUSAGES plan is not locked into either paradigm and provides extensive support for both. This allows planners to easily generate plans that can both retraverse known territory using map based links and explore new territory using event based links, assuming that event detectors can be built.

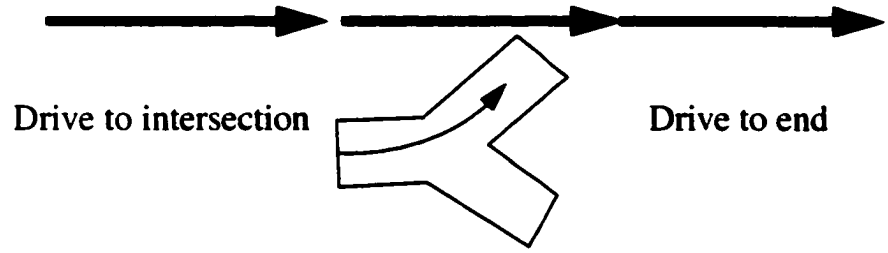


Figure 49 Mixing geometric and cognitive links

In order to integrated Annotated Maps with SAUSAGES we combined the abilities of the Navigator and the Map Manager into one module. This module serves as the maintainer of the global to local position transform, the source for geometric information of the world, and the resource for active queries such as trigger detection. We extended the active queries to be more

general than simply the vehicle crossing a line, and can now do arbitrary active queries, such as respond to the vehicle entering an area, or crossing a line in a particular direction.

SAUSAGES' Link Manager can use this extended Annotated Maps module as a resource for geometric events which will cause link transitions. Other modules in the system can use the extended Annotated Maps manager to report changes in the vehicle's global position estimate as well as a source of geometric information.

7.7 Conclusions

Annotated maps are an example of a tool which enables our robot systems to perform tasks which would be otherwise impossible given the current level of ability of our perception systems. They were not developed to mimic humans, or for their own sake, they were developed purely in the response to a need in our mobile robot systems. They transform the difficult problem of detecting features in the world and reacting to them to the more tractable problem of monitoring the vehicle's position and reacting to that. Further, annotated maps guide and support perception by providing advice about what to look for and when and where to look for it.

Annotated maps allowed us to perform long autonomous missions without having to develop specialized event detectors for transitioning between the stages of a plan. They give us the freedom to only use the perception that is needed to move the vehicle anyway to overcome the inevitable drifting in vehicle position. By combining the rigid map based representation with an event based plan execution and monitoring system, we can overcome the inflexibilities of a pure annotated map representation and perform a wide variety of tasks with existing hardware and software. We do not have to limit ourselves to simulation, where features and events can be easily "detected" and we do not have to wait for competent generic feature detectors: we can use Annotated Maps and SAUSAGES to execute complex, useful, and interesting missions right now in the real world.

Chapter 8 Conclusion

8.1 Contributions

This thesis flows from years of practical work in developing outdoor mobile robot systems. Over the years these systems have gone from driving our original Navlab slowly along a park path to steering a minivan across the country at highway speeds. The systems have gone from simple, single modalities such as local obstacle detection to performing complex missions involving several cooperating vehicles using obstacle avoidance, strategic planners, teleoperation and road following. The development of these systems has not only happened at the Carnegie Mellon Robotics Institute. Many of the tools, components, and philosophies in the Navlab group's systems have been ported to Lockheed-Martin's part of the Unmanned Ground Vehicle program's effort to show real vehicles performing militarily relevant tasks such as scouting, reconnaissance, and target acquisition.

In wrestling with the issues of building real systems to perform real tasks on an outdoor mobile robot, I have built several concrete tools.

- IPT - an object oriented interprocess communications toolkit. IPT is the primary architectural infrastructure of my systems, i.e., it is the main mechanism by which modules interact and communicate. IPT provides simple, consistent point to point communications between the modules of the system, regardless of CPU type or underlying transport mechanisms. For example, the process of sending a message between modules running

on a Sun workstation and a PC will be the same as the process of sending a message between two processes running on a real time operating system, even though in the first case IPT must switch the byte order of data and use TCP/IP while in the second case IPT simply uses shared memory to transfer the messages. IPT has been tuned to support the module oriented approach to system integration I use in building my robot systems.

- SAUSAGES - a general plan specification, monitoring, and execution system. SAUSAGES provides system integrators with a tool to quickly prototype simple missions, and build on those simple missions to work in more complex scenarios. SAUSAGES lets a system integrator build up a "vocabulary" for the system which will translate abstract instructions, such as "follow the road," or "teleoperate while avoiding obstacles" into all of the low level details of what modules have to be started with what empirically discovered parameters. With SAUSAGES, a system integrator can easily build and tweak simple sequences of actions to fine tune the transitions between plan steps. The resulting vocabulary can be used by the system integrator, graphical user interfaces, or even reasoning systems to specify complex missions that will work in the real world on a real vehicle.

- Annotated maps - a map based plan execution scheme. Our perception systems cannot robustly detect arbitrary events in the unstructured, noisy, and unpredictable environment that exists outside of laboratories, but our positioning systems are fairly reliable. Annotated maps convert the difficult problem of looking for events in the environment to trigger plan actions to the more tractable problem of monitoring the vehicle position. Thus annotated maps are a tool that acknowledges our current system's weakness and takes advantage of its strong points to execute plans and scenarios that would otherwise be far out of reach.

I am not saying these tools should be the basis of all mobile robot systems for all time. I have developed these tools to address our current task requirements, organizational ambitions, and algorithmic capabilities. In a few years our requirements, ambitions, and capabilities may change, and then these tools may have to adapt, expand, or be discarded as obsolete. In fact, different

groups may currently have different requirements, ambitions, and capabilities that would require different infrastructure and planning tools.

Beyond the specific software constructs, the real contribution of this thesis is to embrace the reality that different requirements, ambitions, and capabilities will be best addressed by different system software architectures. A system software architecture covers the software infrastructure, i.e. how modules communicate; the software content, i.e. what modules are included in the system; and the software topology, i.e. what modules are connected to what modules and what data flows between them. If requirements, ambitions, and capabilities change, then there is a good chance that some aspect of the system software architecture will have to be changed to best address the new situation. Often the changes will simply involve putting new modules into well defined slots in the architecture, but sometimes the changes will involve changes in the architectural paradigms and standards themselves.

Mobile robotics is a rapidly changing and evolving field, and all of the components of an outdoor mobile robot system, from the sensors to the software architecture, will undergo continuous research and development. It is impossible at the beginning of a project to predict the requirements and capabilities of perception modules being developed over the course of the project. Changes in these requirements and capabilities will be reflected by changes in ambitions. Thus, the overall software architecture cannot be fixed at the beginning of the design process, but must be free to emerge from the actual requirements and capabilities of the system as they are discovered through experimentation and development. Thus, systems depend on a fluid, emergent architecture rather than a fixed reference standard.

The emergent architecture philosophy would be useless without the tools to support it. In order to free an architecture to emerge from the real requirements and capabilities as they are discovered, I have developed a reconfigurable interface system. Reconfigurable interfaces isolate individual modules, such as road followers, command arbiters, or reasoning systems, from the architecture in which they are embedded. Reconfigurable interfaces fool the modules, and the module developers, into thinking that the module is at the center of the system. The rest of the system is hidden behind abstract data sources, such as sources for images or vehicle positions, and destinations, such as consumers of vehicle commands or semantic information.

Reconfigurable interfaces allow components to be shared across projects with different software architectures and to easily roll with the inevitable changes in a software architecture that happen over a period of time in a single project. In one configuration, the module will exist in a stand alone architecture, in which it is directly driving the vehicle, while in another configuration, the module will exist in an integrated architecture as one agent moving the vehicle under the direction of plan supervisors. A system integrator can experiment with different architectures, or move the module to a different architecture used by a different group simply by adding new interface configurations. The module's view of the system remains constant, even as the system around it changes.

Reconfigurable interfaces can also be used as debugging and development tools. There can be interface configurations for using simulated data, for recording data, and for using recorded data in a playback mode. These features of reconfigurable interfaces ease the integration of a new module into an existing system by reducing and formalizing the interaction between system integrators and module developers. System integrators will find real world situations which induce bugs in algorithms. They can then use reconfigurable interfaces to capture those situations and replay them, and then tell the module developers what parameters and data to give to stand alone playback interfaces in order to reproduce the bug. The module developers can then fix the bug without having to deal with all of the intricacies of the system, while the system integrators get the bug fixed without having to deal with all of the intricacies of the module.

8.2 Limitations and scope

I developed the emergent architectures approach in response to the pressures of developing large mobile robot systems whose development was distributed across academic and industrial sites. These pressures primarily come from two conflicting sources:

1. The individual developers have their own agendas, ideologies and approaches, and cannot be easily dictated to. They would like to view their modules are research vehicles which change as new information is discovered and new avenues of research open up.

2. The tasks and architectures are evolving at the same rate as the individual modules, and the individual modules are large and complex as well as being dynamic, so reimplementing each module, or even simply having a different version of them for every different task and architecture is an overwhelming burden.

Reconfigurable interfaces are an attempt to compromise in the conflict between module developers, who want stable architectures in which their modules can reside, and system integrators, who want stable modules to build and experiment with different architectures and tasks.

Reconfigurable interfaces are a compromise: they are not a panacea for all system integration ills. Module developers know best what resources their algorithms need and the representation that best captures all of the information their algorithm produces. The system integrators know what information the current system can produce and the best representation for the system of an algorithm's output. Reconfigurable interfaces give a consistent location to translate between these two requirements, and thus give the system integrator with a consistent view of the module as it changes, and give the module developer a consistent view of the system as it changes. The problems come when a module developer demands a resource that the system integrator cannot provide, or the system demands information that cannot be extracted from the module. At that point, collaborative work must be done to change the reconfigurable interfaces and either the system or the module to fit the new requirements.

There are costs of using reconfigurable interfaces. First, there is the practical matter of the translations that take place within the reconfigurable interfaces which result in some cost to the efficiency of the system, although in the reconfigurable interfaces implementation I have endeavored to keep this translation cost minimal. More fundamentally, by hiding the details of the system architecture from a module developer, we build a loosely coupled system. If the module developer knew all the details of a whole system, then that developer could possibly build a very tightly coupled system by taking advantages of the knowledge of the system structure. This is the same trade-off mode in any large software system. i.e. module developers give up having to know about the details of the system in order to make module development more tractable, but then some of the potential functionality of the system is hidden from the module developer's view.

I contend that the utility of reconfigurable interfaces is not limited to outdoor mobile robot systems in which academic research is involved. Imagine a system integration environment in which the system integrator had absolute control over the module developers and could force them to accept any architectural standards that are proposed. Even when such standards can be imposed upon a system, they cannot be imposed upon nature. If the components of a system are

areas of research themselves, their requirements and capabilities cannot be completely predicted, regardless of how much a system integrator wishes it to be so, and so any a priori standards enforced upon them before their true requirements and capabilities are discovered could prove to be a serious burden.

Thus I believe that the emergent architecture approach can be generalized beyond the outdoor mobile robot domain to any domain which is undergoing radical change through research and development. The more unstable and uncertain the components of a system are, the more the system will benefit from the emergent architecture approach. Conversely, the need for the emergent architecture approach diminishes as the domain becomes more strictly defined and explored. Eventually, a domain may be explored thoroughly enough so that there is a stable set of components and a stable family of reference architectures in which those components can be combined to perform interesting missions. The emergent architecture approach provides a means for getting to that developmentally stable plateau. Emergent architectures' reconfigurable interfaces provide a mechanism for exploring the architectural options and adapting to new information while reusing components and easing the transitions as much as possible.

8.3 Future Work and Vision

At present I have only applied reconfigurable interfaces, the tools of an emergent architecture, to a small segment of our systems. Our road follower, ALVINN, uses reconfigurable interfaces so that it can be moved from a dedicated road following task to an integrated multi-modal task. D*, a dynamic global trajectory planner also uses reconfigurable interfaces to demonstrate how reconfigurable interfaces ease the development and integration process for an individual module in an individual task.

Having only a small percentage of system modules integrated using reconfigurable interfaces actually shows a great advantage of this approach to emergent architectures: using reconfigurable interfaces is not an all or nothing proposition. The concepts and tools of emergent architectures can be incrementally introduced into an existing system rather than requiring a complete overhaul in order to be effective. But, the more modules that use the reconfigurable interfaces, the more flexible the architecture will be.

The ultimate goal is to investigate the effect of using the philosophies and tools of the emergent architecture approach as the basis of an entire system rather than simply something added in midstream of the development. My vision is to revamp the way outdoor mobile robot systems are designed.

All too often the software design meeting for an outdoor mobile robot system is a frustrating, draining, tension-filled event. All of the people involved in building the system are gathered into a room to decide how the system should be structured. The tension comes from the understood purpose of the meeting: to design standards that will last the lifetime of the project. These standards will dictate and regulate how each module interacts with the rest of the system, the sensors, and the world. Everyone in the room knows that if the standards are incomplete, or, worse, based on incorrect assumptions, there will be dire consequences later in the project. Incomplete or erroneous standards mean a choice has to be made between either living with a software architecture that is flawed and results in suboptimal performance, or changing the standards which permeate the system. Either alternative is at least inconvenient, and worse, can be disastrous for the schedule of development. Therefore, there is intense pressure to get the standards "right" from the beginning.

For most outdoor mobile robot projects, this search for **the** set of architectural standards is at best futile and at worst damaging. Since so much work in an outdoor mobile robot project will be unprecedented and developmentally unstable, there is very little hope that the roomful of designers will be able to predict what the actual task requirements of the system are and what the various components of the system will actually be able to do. For example, on the pessimistic side, there is the tendency of perception researchers to be over-optimistic about what future versions of their modules will be able to detect reliably in the actual task environment. On the optimistic side there is the possibility of the development of a radically new algorithm or approach which has the potential to positively impact the system's performance, but which will not fit well in the original architectural paradigm. Any set of standards decided at this initial design meeting will be either too general to be useful or too restrictive to be able to react to the inevitable unexpected developments.

When a group using the emergent architecture philosophy has an initial design meeting, there should be much less tension and apprehension. The flow of such a meeting should be as follows:

- What functions need to be performed? How will those be grouped as modules?
- What modules need to be developed or integrated for the next mission we need to demonstrate? What needs to be perceived? What kind of reactions do there have to be? What kind of planning does there need to be? Is a reasoning system necessary?
- Who will develop each module?
- What does each module need as input? What does it produce as output?
- Are there commonalities across modules so we can reduce the number of reconfigurable interfaces?

This initial design process is less stressful because there are no far reaching, system wide standards being decided. There is no commitment to a system architecture at all. At this point a "standard" is being specified for each particular module's view of the system, so that the system integrators can quickly create reconfigurable interfaces that adhere to those standards and which the module developers can use immediately for stand alone research and development. Straw men architectures are useful at this point solely for the purpose of identifying commonalities in interfaces between modules and for giving general direction to the search for what functionalities are required to make the system perform the given task. These architectures are understood to be merely guesses, and the expectation, not the fear, is that the real system architecture will emerge as the system integrators build, integrate, and test the whole system operating on a real task in the real world.

The emergent architecture approach has this potential for radical impact on the design process because it acknowledges the unfortunate facts of working in a young, dynamic field like outdoor mobile robot systems. The emergent architecture approach is not to predict what a system might need to succeed, but rather to build mechanisms into the development and integration process which enable reaction and adaptation to what the system does need to succeed.

References

Agre, P. E. and Chapman, D. (1987). PENGI: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, Menlo Park, CA. American Association for Artificial Intelligence.

Agre, P. E. and Chapman, D. (1990). What are plans for? *IEEE Transactions on Robotics and Automation*, 6:17–34.

Albus, J. S., McCain, H. G., and Lumia, R. (1987). NASA/NBS standard reference model for telerobot control. Technical Report 1235, NBS.

Amidi, O. (1990). Integrated mobile robot control. Technical report, Robotics Institute, Carnegie Mellon University.

Arkin, R. C. (1989). Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112.

Bares, J., Hebert, M., Kanade, T., Krotkov, E., Mitchell, T., Simmons, R., and Whittaker, W. (1989). Ambler: An autonomous rover for planetary exploration. *IEEE Computer*.

Berry, G. and Gonthier, G. (1992). The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152.

Bhaskar, K. (1983). How object-oriented is your system? *SIGPLAN Notices*, 18(10).

Biggerstaff, T. and Perlis, A. (1989). *Software Reusability*. ACM Press.

Booch, G. (1991). *Object Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA.

Borenstein, J. and Koren, Y. (1991). The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Journal of Robotics and Automation*, 7(3):278–288.

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

Brooks, R. A. (1991). Intelligence without reason. *International Journal of Computing and Artificial Intelligence*.

Carbonell, J. G. and Veloso, M. M. (1988). Integrating derivational analogy into a general problem solving architecture. In Kolodner, J., editor, *Proceedings of the First Workshop on Case Based Reasoning*.

Coombs, C., Raiffa, H., and Thrall, R. (1954). Some views on mathematical models and measurement theory. *Psychological Review*, 61(2).

Crisman, J. D. and Thorpe, C. E. (1990). Color vision for road following. In Thorpe, C. E., editor, *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 2. Kluwer Academic Publishers.

Elfes, A. (1987). Sonar-based real-world mapping and navigation. *IEEE Journal of Robotics and Automation*, 3(3):249–265.

Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). KQML as an agent communication language. In *Proc. of the Third International Conference on Information and Knowledge Management*, pages 456–463, Gaithersburg, MD. ACM.

Firby, R. (1989). Adaptive execution in complex dynamic worlds. Technical Report YALEU/CSD/RR #672, Yale University.

Gat, E. (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings Tenth National Conference on Artificial Intelligence*, pages 809–15, San Diego. AAAI.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts.

Georgeff, M. P. and Lansky, A. L. (1986). Procedural knowledge. In *Proc. IEEE Special Issue on Knowledge Representation*, pages 1383–1398.

Gowdy, J. (1994). Sausages: Between planning and action. Technical Report CMU-RI-TR-94-32, Robotics Institute, Carnegie Mellon.

Gowdy, J. (1996). IPT: An object oriented toolkit for interprocess communication. Technical Report CMU-RI-TR-96-07, Robotics Institute, Carnegie Mellon.

Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, 26:pp. 251–321.

Hayes-Roth, F. (1994). Architecture-based acquisition and development of software: Guidelines and recommendations from the ARPA Domain-Specific Software Architecture (DSSA) program. Technical report, Teknowledge.

Hebert, M., Kweon, I., and Kanade, T. (1990). 3-d vision techniques for autonomous vehicles. In Thorpe, C. E., editor, *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 8. Kluwer Academic Publishers.

Hexmoor, H., Lammens, J., and Shapiro, S. (1993). An autonomous agent architecture for integrating unconscious and conscious, reasoned behaviors. *Computer Architectures for Machine Perception*.

Huhns, M. N. and Singh, M. P. (1995). A mediated approach to open, large-scale information management. In *Proc. of the 1995 IEEE Fourteenth Annual International Phoenix Conference on Computers and Communications*, pages 115–121, Scottsdale, AZ.

Jochem, T., Pomerleau, D., Kumar, B., and Armstrong, J. (1995). PANS: A portable navigation platform. In *1995 IEEE Symposium on Intelligent Vehicles*, Detroit, Michigan, USA.

Kluge, K. and Thorpe, C. (1990). Explicit models for robot road following. In Thorpe, C. E., editor, *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 3. Kluwer Academic Publishers.

Kortenkamp, D. and Weymouth, T. (1994). Topological mapping for mobile robots using a combination of sonar and vision sensing. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 979–984, Menlo Park.

Kuipers, B. and Levitt, T. (1988). Navigation and mapping in large-scale space. *AI Magazine*.

Laird, J., Newell, A., and Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1).

Langer, D., Rosenblatt, J., and Hebert, M. (1994). An integrated system for autonomous off-road navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 61–72, San Diego. IEEE.

Luchins, A. S. (1942). Mechanization in problem solving. *Psychological Monographs*, 54(248).

M. P. Georgeff, A. L. Lansky, M. J. S. (1987). Reasoning and planning in dynamic domains: An experiment with a mobile robot. Technical Report 380, SRI.

M. W. Gertz, D. B. Stewart, P. K. K. (1994). A human machine interface for distributed virtual laboratories. *IEEE Robotics an Automation Magazine*, 1(4):5–13.

Mataric, M. J. (1990). A distributed model for mobile robot environment-learning and navigation. Master's thesis, MIT, Cambridge, MA.

Message Passing Interface Forum (1994). MPI: A message-passing interface standard. Technical Report CS-94-230, University of Tennessee.

Mettala, E. and Graham, M. (1992). The domain-specific software architecture program. Technical Report CMU/SEI-92-TR-22 ESD-92-TR-223, Carnegie Mellon Software Engineering Institute.

Moravec, H. P. (1988). Sensor fusion in certainty grids for mobile robots. *AI Magazine*, pages 61–74.

Munkeby, S. and Spofford, J. (1996). UGV/Demo II program: Status through Demo C and Demo II preparations. In *Proc. 10th Annual International AeroSense Symposium*.

Newell, A. and Simon, H. A. (1963). GPS, a program that simulates human thought. In Feigenbaum, E. A. and Feldman, J., editors, *Computers and Thought*. McGraw-Hill.

Nilsson, N. (1984). Shakey the robot. Technical Report 323, SRI, Menlo Park, CA.

Object Management Group (1993). *The Common Object Request Broker: Architecture and Specification*. The Object Management Group.

Payton, D. (1986). An architecture for reflexive autonomous vehicle control. In *IEEE Conference on Robotics and Automation*, pages 1838–1845, San Francisco, CA.

Pomerleau, D. (1989). ALVINN: An autonomous land vehicle in a neural network. In Touretzky, D., editor, *Advances in Neural Information Processing Systems I*. Morgan Kaufmann.

Pomerleau, D. and Jochem, T. (1996). Rapidly adapting machine vision for automated vehicle steering. *IEEE Expert*, 11(2).

Rosenblatt, J. and Payton, D. (1989). A fine-grained alternative to the subsumption architecture for mobile robot control. In *Proc. of the IEEE/INNS International Joint Conference on Neural Networks*, pages 317–324, Washington DC. IEEE.

Rosenschein, S. and Kaelbling, L. (1986). The synthesis of digital machines with provable epistemic properties. In *Proc. Theoretical Aspects of Reasoning about Knowledge*, pages 83–98, San Diego.

Rosenschein, S. J. (1985). Formal theories of knowledge in artificial intelligence. Technical Report 362, SRI.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, pages 1039–1046.

Simmons, R. (1994). Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43.

Simon, H. and Siklossy, L. (1972). *Representation and Meaning*. Prentice-Hall, Englewood Cliffs, NJ.

Stentz, A. (1990). The CODGER system for mobile robot navigation. In Thorpe, C., editor, *Vision and Navigation: The Carnegie Mellon Navlab*, chapter 9. Kluwer Academic Publishers.

Stentz, A. (1994). Optimal and efficient path planning for partially known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Taylor, R. H., Summers, P. D., and Meyer, J. M. (1983). AML: A manufacturing language. *International Journal of Robotics Research*, 1(3):19–41.

Thorpe, C. (1990). *Vision and Navigation: the The Carnegie Mellon Navlab*. Kluwer Academic Publishers.

Thorpe, C., Amidi, O., Gowdy, J., Hebert, M., and Pomerleau, D. (1991). Integrating position measurement and image understanding for autonomous vehicle navigation. In *Second International Workshop on High Precision Navigation*.

Thorpe, C. and Gowdy, J. (1990). Annotated maps for autonomous land vehicles. In *Proceedings of DARPA Image*

Understanding Workshop, Pittsburgh PA.

Thrun, S. (1998). Learning metric-topological maps for indoor mobile robot navigation. *AI Journal*, 99(1):21–71.

VAL II (1983). *User's Guide to VAL II, Preliminary Program Documentation, ver. X2*. Unimation, Inc., Danbury, Conn.

Wiederhold, G. (1992). Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49.