8. Robotic Systems Architectures and Programming

David Kortenkamp, Reid Simmons

Robot software systems tend to be complex. This complexity is due, in large part, to the need to control diverse sensors and actuators in real time, in the face of significant uncertainty and noise. Robot systems must work to achieve tasks while monitoring for, and reacting to, unexpected situations. Doing all this concurrently and asynchronously adds immensely to system complexity.

The use of a well-conceived architecture, together with programming tools that support the architecture, can often help to manage that complexity. Currently, there is no single architecture that is best for all applications – different architectures have different advantages and disadvantages. It is important to understand those strengths and weaknesses when choosing an architectural approach for a given application.

This chapter presents various approaches to architecting robotic systems. It starts by defining terms and setting the context, including a recounting of the historical developments in the area of robot architectures. The chapter then discusses in more depth the major types of architectural components in use today – behavioral control (Chap. 38), executives, and task planners (Chap. 9) – along with commonly used techniques for interconnecting those components. Throughout, emphasis will be placed on programming tools and environments that support these architectures. A case study is then presented, followed by a brief discussion of further reading.

8.1 Overview

The term *robot architecture* is often used to refer to two related, but distinct, concepts. Architectural *structure* refers to how a system is divided into subsystems and how those subsystems interact. The structure of a robot system is often represented informally using traditional *boxes and arrows* diagrams or more formally using techniques such as unified modeling language (UML) [8.1]. In contrast, architectural *style* refers to the computational concepts that underlie a given system. For instance, one robot system might use a publish–subscribe message passing style of communication, while another may use a more synchronous client–server approach.

All robotic systems use some architectural structure and style. However, in many existing robot systems it is difficult to pin down precisely the architecture being used. In fact, a single robot system will often use several styles together. In part, this is because the system implementation may not have clean subsystem bound-

aries, blurring the architectural structure. Similarly, the architecture and the domain-specific implementation are often intimately tied together, blurring the architectural style(s).

This is unfortunate, as a well-conceived, clean architecture can have significant advantages in the specification, execution, and validation of robot systems. In general, robot architectures facilitate development by providing beneficial constraints on the design and implementation of robotic systems, without being overly restrictive. For instance, separating behaviors into modular units helps to increase understandability and reusability, and can facilitate unit testing and validation.

8.1.1 Special Needs of Robot Architectures

In some sense, one may consider robot architectures as software engineering. However, robot architectures are distinguished from other software architectures because of the special needs of robot systems. The most important of these, from the architectural perspective, are that robot systems need to interact asynchronously, in real time, with an uncertain, often dynamic, environment. In addition, many robot systems need to respond at varying temporal scopes – from millisecond feedback control to minutes, or hours, for complex tasks.

To handle these requirements, many robot architectures include capabilities for acting in real time, controlling actuators and sensors, supporting concurrency, detecting and reacting to exceptional situations, dealing with uncertainty, and integrating high-level (symbolic) planning with low-level (numerical) control.

While the same capability can often be implemented using different architectural styles, there may be advantages of using one particular style over another. As an example, consider how a robot system's style of communications can impact on its reliability. Many robot systems are designed as asynchronous processes that communicate using message passing. One popular communication style is *client–server*, in which a message request from the client is paired with a response from the server. An alternate communication paradigm is *publish–subscribe*, in which messages are broadcast asynchronously and all modules that have previously indicated an interest in such messages receive a copy. With client–server-style message passing, modules typically send a request and then block, waiting for the response. If the response never comes (e.g., the server module crashes) then deadlock can occur. Even if the module does not block, the control flow still typically expects a response, which may lead to unexpected results if the response never arrives or if a response to some other request happens to arrive first. In contrast, systems that use publish–subscribe tend to be more reliable: because messages are assumed to arrive asynchronously, the control flow typically does not assume any particular order in which messages are processed, and so missing or out-of-order messages tend to have less impact.

8.1.2 Modularity and Hierarchy

One common feature of robot architectures is modular decomposition of systems into simpler, largely independent pieces. As mentioned above, robot systems are often designed as communicating processes, where the communications interface is typically small and relatively low bandwidth. This design enables the processes/modules to handle interactions with the environment asynchronously, while minimizing interactions with one another. Typically, this decreases overall system complexity and increases overall reliability.

Often, system decomposition is hierarchical – modular components are themselves built on top of other modular components. Architectures that explicitly support this type of layered decomposition reduce system complexity through abstraction. However, while hierarchical decomposition of robotic systems is generally regarded as a *desirable quality*, debate continues over the dimensions along which to decompose. Some architectures [8.2] decompose along a temporal dimension – each layer in the hierarchy operates at a characteristic frequency an order of magnitude slower than the layer below. In other architectures [8.3–6], the hierarchy is based on task abstraction – tasks at one layer are achieved by invoking a set of tasks at lower levels. In some situations, decomposition based on spatial abstraction may be more useful, such as when dealing with both local and global navigation [8.7]. The main point is that different applications need to decompose problems in different ways, and architectural styles can often be found to accommodate those different needs.

8.1.3 Software Development Tools

While significant benefit accrues from *designing* systems using well-defined architectural styles, many architectural styles also have associated software tools that facilitate adhering to that style during *implementation*. These tools can take the form of libraries of functions calls, specialized programming languages, or graphical editors. The tools make the constraints of the

architectural style explicit, while hiding the complexity of the underlying concepts.

For instance, inter-process communication libraries, such as common object request broker architecture (CORBA) [8.8] and inter-process communication (IPC) package [8.9], make it easy to implement message passing styles, such as client–server and publish–subscribe, respectively. Languages, such as Subsumption [8.10] and Skills [8.11] facilitate the development of data-driven, real-time behaviors, while languages such as the execution support language (ESL) [8.12] and the the plan execution interchange language (PLEXIL) [8.13] provide support for reliably achieving higher-level tasks. Graphical editors, such as found in ControlShell [8.14], Labview [8.15] and open robot controller computer aided design (ORCCAD) [8.6], provide constrained ways to assemble systems, and then automatically generate code that adheres to the architectural style.

In each case, the tools facilitate the development of software in the given style and, more importantly, make it impossible (or, at least, very difficult) to violate the constraints of that architectural style. The result is that systems implemented using such tools are typically easier to implement, understand, debug, verify, and maintain. They also tend to be more reliable, since the tools provide well-engineered capabilities for commonly needed control constructs, such as message passing, interfacing with actuators and sensors, and handling concurrent tasks.

8.2 History

Robot architectures and programming began in the late 1960s with the Shakey robot at Stanford University [8.16] (Fig. 8.1). Shakey had a camera, a range finder, and bump sensors, and was connected to DEC PDP-10 and PDP-15 computers via radio and video links. Shakey's architecture was decomposed into three functional elements: sensing, planning, and executing [8.17]. The sensing system translated the camera image into an internal world model. The planner took the internal world model and a goal and generated a plan (i. e., a series of actions) that would achieve the goal. The executor took the plan and sent the actions to the robot. This approach has been called the *sense–plan–act* (*SPA*) paradigm (Fig. 8.2). Its main architectural features are that sensing flowed into a world model, which was then used by the planner, and that plan was executed without directly using the sensors that created the model. For many years, robotic control architectures and programming focused almost exclusively on the SPA paradigm.

Fig. 8.1 Shakey (courtesy sri.com)

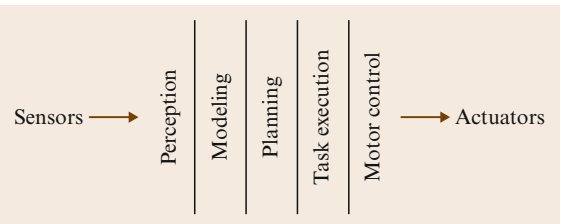

Fig. 8.2 The sense–plan–act (SPA) paradigm (after [8.3], with permission)

8.2.1 Subsumption

In the early 1980s, it became apparent that the SPA paradigm had problems. First, planning in any real-world domain took a long time, and the robot would be blocked, waiting for planning to complete. Second, and more importantly, execution of a plan without involving sensing was dangerous in a dynamic world. Several new robot control architecture paradigms began to emerge, including reactive planning, in which plans were generated quickly and relied more directly on sensed information instead of internal models [8.4, 18]. The most influential work, however, was the subsumption architecture of *Brooks* [8.3]. A subsumption architecture is built from layers of interacting finite-state machines – each connecting sensors to actuators directly (Fig. 8.3). These finite-state machines were called

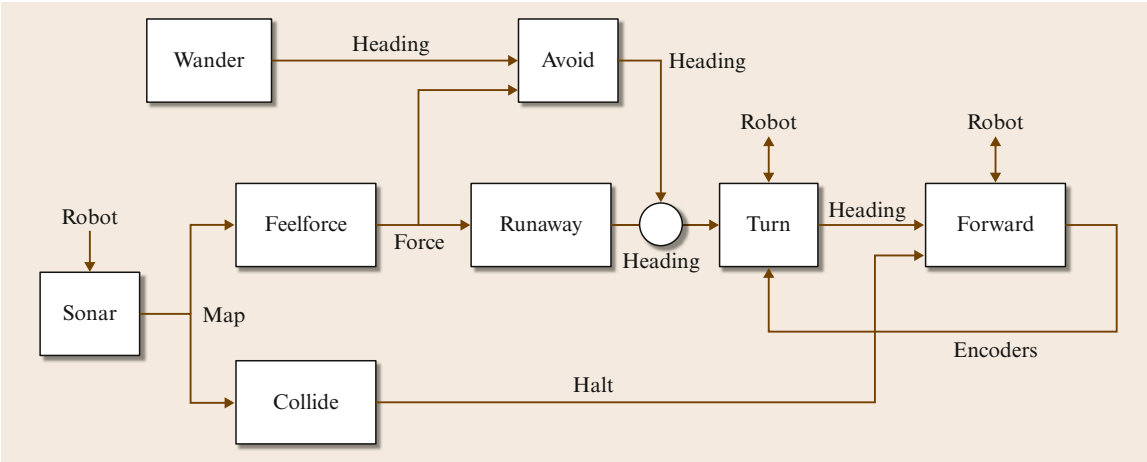

Fig. 8.3 Example of the Subsumption architecture (after [8.3], with permission)

behaviors (leading some to call Subsumption *behavior-based* or *behavioral* robotics [8.19]; see also Ch. 38). Since multiple behaviors could be active at any one time, Subsumption had an *arbitration* mechanism that enabled higher-level behaviors to override signals from lower-level behaviors. For example, the robot might have a behavior that simply drives the robot in random directions. This behavior is always active and the robot is always driving somewhere. A second, higher-level behavior could take sensor input, detect obstacles, and steer the robot away from them. It is also always active. In an environment with no obstacles, the higher-level behavior never generates a signal. However, if it detects an obstacle it overrides the lower-level behavior and steers the robot away. As soon as the obstacle is gone (and the higher-level behavior stops sending signals), the lower-level behavior gets control again. Multiple, interacting layers of behaviors could be built to produce more and more complex robots.

Many robots were built using the subsumption approach – most at MIT [8.20–22]. They were quite successful. Whereas SPA robots were slow and ponderous, Subsumption robots were fast and reactive. A dynamic world did not bother them because they constantly sensed the world and reacted to it. These robots scampered around like insects or small rodents. Several behavioral architectures arose in addition to Subsumption, often with different arbitration schemes for combining the outputs of behaviors [8.23, 24].

A popular example of behavior-based architectures is *Arkin*'s motor-control schemas [8.25]. In this biologically inspired approach, motor and perceptual schemas [8.26] are dynamically connected to one another. The motor schemas generate response vectors based on the outputs of the perceptual schemas, which are then combined in a manner similar to potential fields [8.27]. To achieve more complex tasks, the autonomous robot architecture (AuRA) architecture [8.28, 29] added a navigation planner and a plan sequencer, based on finite-state acceptors (FSAs), to the reactive schemas.

However, behavior-based robots soon reached limits in their capabilities. It proved very difficult to compose behaviors to achieve long-range goals and it proved almost impossible to optimize robot behavior. For example, a behavior-based robot that delivered mail in an office building could easily be built by simply wandering around the office building and having behaviors looking for rooms and then overriding the wandering and entering the office. It was much more difficult to use the behavioral style of architecture to design a system that reasoned about the day's mail to visit the offices in an optimal order to minimize delivery time. In essence, robots needed the planning capabilities of the early architectures wedded to the reactivity of the behavior-based architectures. This realization led to the development of layered, or tiered, robot control architectures.

8.2.2 Layered Robot Control Architectures

One of the first steps towards the integration of reactivity and deliberation was the reactive action packages (RAPs) system created by *Firby*. In his thesis [8.30], we see the first outline of an integrated, three-layer architecture. The middle layer of that architecture, and the subject of the thesis, was the RAPs system. Firby also

speculated on the form and function of the other two tiers, specifically with the idea of integrating classic deliberative methods with the ideas of the emerging situated reasoning community, but those layers were never implemented. Later, *Firby* would integrate RAPs with a continuous low-level control layer [8.31].

Independently and concurrently, *Bonasso* at MITRE [8.32] devised an architecture that began at the bottom layer with robot behaviors programmed in the Rex language as synchronous circuits [8.33]. These Rex machines guaranteed consistent semantics between the agent's internal states and those of the world. The middle layer was a conditional sequencer implemented in the Gapps language [8.34], which would continuously activate and deactivate the Rex skills until the robot's task was complete. This sequencer based on Gapps was appealing because it could be synthesized through more traditional planning techniques [8.35]. This work culminated in the 3T architecture (named after its three tiers of interacting control processes – planning, sequencing, and real-time control), which has been used on many generations of robots [8.36].

Architectures similar to 3T (Fig. 8.4) have been developed subsequently. One example is ATLANTIS [8.37], which leaves much more control at the sequencing tier. In ATLANTIS, the deliberative tier must be specifically called by the sequencing tier. A third example is *Saridis*' intelligent control architecture [8.38]. The architecture begins with the servo systems available on a given robot and augments them to integrate the execution algorithms of the next level, using VxWorks and the VME bus. The next level consists of a set of coordinating routines for each lower subsystem, e.g., vision, arm motion, and navigation. These are implemented in Petri net transducers (PNTs), a type of scheduling mechanism, and activated by a dispatcher connected to the organizational level. The organizational level is a planner implemented as a Boltzmann neural network. Essentially the neural network finds a sequence of actions that will match the required command received as text input, and then the dispatcher executes each of these steps via the network of PNT coordinators.

The LAAS architecture for autonomous systems (LAAS) is a three-layered architecture that includes software tools to support development/programming at each layer [8.39]. The lowest layer (functional) consists of a network of *modules*, which are dynamically parameterized control and perceptual algorithms. Modules are written in the generator of modules (GenoM) language, which produces standardized templates that facilitate the integration of modules with one another.

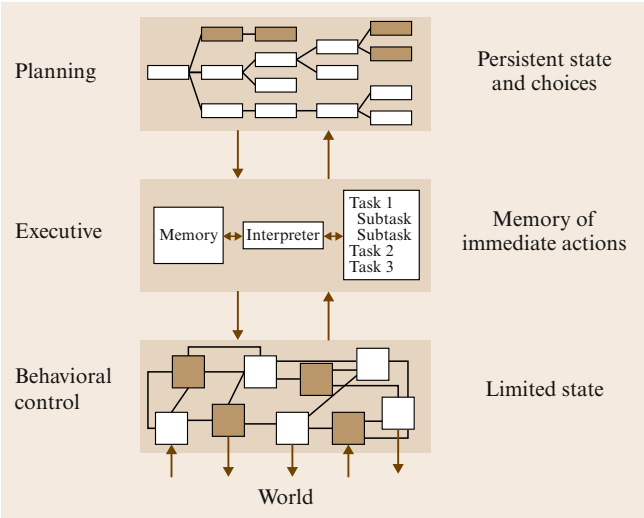

Fig. 8.4 Prototype three-tiered architecture

Unlike most other three-layered architectures, the executive layer is fairly simple – it is purely reactive and does no task decomposition. It serves mainly as a bridge – receiving task sequences from the highest layer and selecting and parameterizing tasks to send to the functional layer. The executive is written in the Kheops language, which automatically generates decision networks that can be formally verified. At the top, the decision layer consists of a planner, implemented using the indexed time table (IxTeT) temporal planner [8.40, 41], and a supervisor, implemented using procedural reasoning system (PRS) [8.42, 43]. The supervisor is similar to the executive layer of other three-layered architectures – it decomposes tasks, chooses alternative methods for achieving tasks, and monitors execution. By combining the planner and supervisor in one layer, LAAS achieves a tighter connection between the two, enabling more flexibility in when, and how, replanning occurs. The LAAS architecture actually allows for multiple decisional layers at increasingly higher levels of abstraction, such as a high-level *mission* layer and a lower-level *task* layer.

Remote agent is an architecture for the autonomous control of spacecraft [8.44]. It actually consists of four layers – a control (behavioral) layer, an executive, a planner/scheduler, and mode identification and recovery (MIR) that combines fault detection and recovery. The control layer is the traditional spacecraft real-time control system. The executive is the core of the architecture – it decomposes, selects, and monitors task execution, performs fault recovery, and does resource management,

turning devices on and off at appropriate times to conserve limited spacecraft power. The planner/scheduler is a batch process that takes goals, an initial (projected) state, and currently scheduled activities, and produces plans that include flexible ranges on start and end times of tasks. The plan also includes a task to reinvoke the planner to produce the next plan segment. An important part of the remote agent is configuration management – configuring hardware to support tasks and monitoring that the hardware remains in known, stable states. The role of configuration management is split between the executive, which uses reactive procedures, and MIR, which uses declarative models of the spacecraft and deliberative algorithms to determine how to reconfigure the hardware in response to detected faults [8.45].

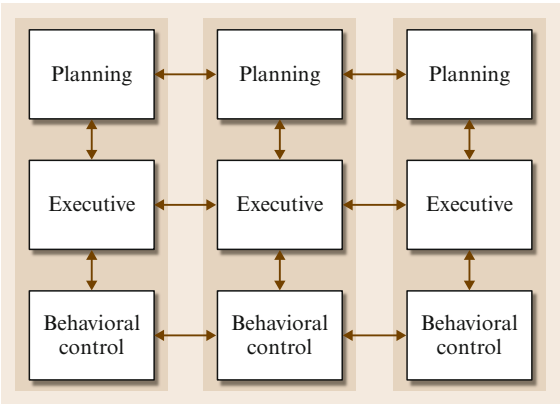

Fig. 8.5 The Syndicate multirobot architecture

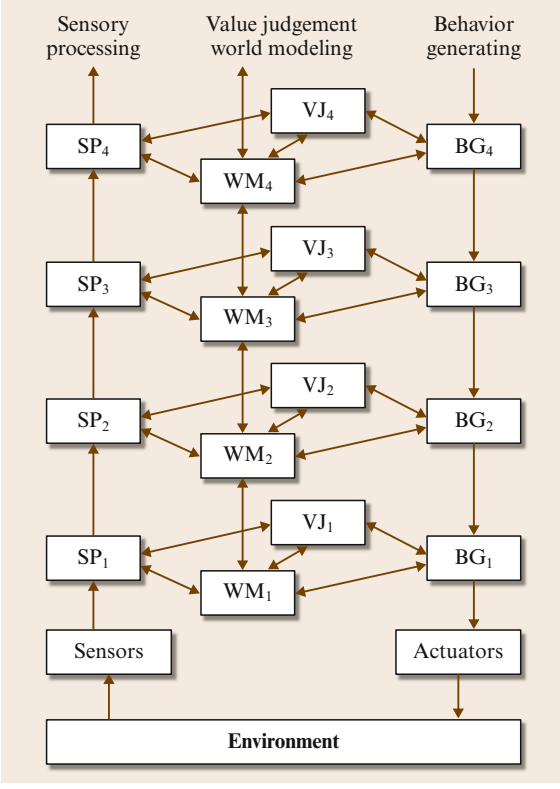

Fig. 8.6 The real-time control system (RCS) reference architecture (after [8.46], with permission)

The Syndicate architecture [8.47] extends the 3T model to multirobot coordination (Chap. 40). In this architecture, each layer interfaces not only with the layers above and below, as usual, but also with the layers of the other robots at the same level (Fig. 8.5). In this way, distributed control loops can be designed at multiple levels of abstraction. The version of Syndicate in [8.48] used a distributed market-based approach for task allocation at the planning layer.

Other noteworthy multitiered architectures have appeared in the literature. The National Bureau of Standards (NBS) developed for the Aeronautics and Space Agency (NASA) the NASA/NBS standard reference model (NASREM) [8.2, 49], later named real-time control system (RCS), was an early reference model for telerobotic control (Fig. 8.6). It is a many-tiered model in which each layer has the same general structure, but operates at increasingly lower frequency as it moves from the servo level to the reasoning levels. With the exception of maintaining a global world model, NASREM, in its original inception, provided for all the data and control paths that are present in architectures such as 3T, but NASREM was a reference model, not an implementation. The subsequent implementations of NASREM followed primarily the traditional sense–plan–act approach and were mainly applied to telerobotic applications, as opposed to autonomous robots. A notable exception was the early work of *Blidberg* [8.50].

While three-layered robot architectures are very popular, various two-layered architectures have been investigated by researchers. The coupled layered architecture for robot autonomy (CLARAty) was designed to provide reusable software for NASA's space robots, especially planetary rovers [8.51, 52]. CLARAty consists of a functional and a decision layer. The functional layer is a hierarchy of object-oriented algorithms that provide more and more abstract interfaces to the robot, such as motor control, vehicle control, sensor-based navigation, and mobile manipulation. Each object provides a generic interface that is hardware independent, so that

the same algorithms can run on different hardware. The decision layer combines planning and executive capabilities. Similar to the LAAS architecture, this is done to provide for tighter coordination between planning and execution, enabling continual replanning in response to dynamic contingencies.

Closed-loop execution and recovery (CLEaR) [8.53] is one instantiation of the CLARAty decision layer. CLEaR combines the continuous activity scheduling, planning, execution and replanning (CASPER) repair-based planner [8.54] and the task description language (TDL) executive language [8.55]. CLEaR provides a tightly coupled approach to goal- and event-driven behavior. At its heart is the capability to do fast, continuous replanning, based on frequent state and resource updates from execution monitoring. This enables the planner to react to many exceptional situations, which can be important in cases where there are many tasks, few resources, and significant uncertainty. In CLEaR, both the planning and executive components are able to handle resource conflicts and exceptional situations – heuristics are used to decide which component should be involved in a given situation. The onboard autonomous science investigation system (OASIS) system [8.56] extends CLEaR to include science data analysis so that the architecture can be driven by opportunistic science-related goals (such as finding unusual rocks or formations). OASIS is planner-centric, releasing tasks to the executive component just a few seconds before their scheduled start times.

The cooperative intelligent real-time control architecture (CIRCA) is a two-layered architecture concerned with guaranteeing reliable behavior [8.57, 58]. It embodies the notion of *bounded reactivity* – an acknowledgement that the resources of the robot are not always sufficient to guarantee that all tasks can be achieved. CIRCA consists of a real-time system (RTS) and an artificial intelligence (AI) system (AIS) that are largely independent. The RTS executes a cyclic schedule of test action pairs (TAPs) that have guaranteed worst-case behavior in terms of sensing the environment and conditionally acting in response. It is the responsibility of the AIS to create a schedule that is guaranteed to prevent catastrophic failures from occurring, while running in hard real time. The AIS does this by planning over a state-transition graph that includes transitions for actions, exogenous events, and the passage of time (e.g., if the robot waits too long, bad things can happen). The AIS tests each plan (set of TAPs) to see if it can actually be scheduled. If not, it alters the planning model, either by eliminating tasks (based on goal prioritization) or by changing parameters of behaviors (e.g., reducing the robot's velocity). The AIS continues this until it finds a plan that can be successfully scheduled, in which case it downloads the new plan to the RTS in an atomic operation.

Like CIRCA, ORCCAD is a two-layered architecture that is concerned with guaranteed reliability [8.6, 59]. In the case of ORCCAD, this guarantee is achieved through formal verification techniques. Robot tasks (lower-level behaviors) and robot procedures (higher-level actions) are defined in higher-level languages that are then translated into the Esterel programming language [8.60], for logical verification, or the Timed-Argus language [8.61], for temporal verification. The verification is geared toward liveness and safety properties, as well as verifying lack of contention for resources.

8.3 Architectural Components

We will take the three-tiered architecture as the prototype for the components discussed in this chapter. Figure 8.4 shows a typical three-tiered architecture. The lowest tier (or layer) is behavioral control and is the layer tied most closely to sensors and actuators. The second tier is the executive layer and is responsible for choosing the current behaviors of the robot to achieve a task. The highest tier is the task-planning layer and it is responsible for achieving long-term goals of the robot within resource constraints. Using the example of an office delivery robot, the behavioral layer is responsible for moving the robot around rooms and hallways, for avoiding obstacles, for opening doors, etc. The executive layer coordinates the behavioral layer to achieve tasks such as leaving a room, going to an office, etc. The task-planning layer is responsible for deciding the order of deliveries to minimize time, taking into account delivery priorities, scheduling, recharging, etc. The task-planning layer sends tasks (e.g., exit the room, go to office 110) to the executive. All these tiers need to work together and exchange information. The next section deals with the problem of connecting components to each other. We then discuss each component of the three-tiered prototype architecture in detail.

8.3.1 Connecting Components

All of the architecture components that have been discussed in this chapter need to communicate with each other. They need to both exchange data and send commands. The choice of how components communicate (often called the *middleware*) is one of the most important and most constraining of the many decisions a robot architecture designer will make. From previous experience, a great deal of the problems and a majority of the debugging time in developing robot architectures have to do with communication between components. In addition, once a communication mechanism is chosen it becomes extremely difficult to change, so early decisions persist for many years. Many developers *roll their own* communication protocols, usually built on top of Unix sockets. While this allows for customization of messages, it fails to take advantage of the reliability, efficiency, and ease of use that externally available communication packages provide. There are two basic approaches to communication: client–server and publish–subscribe.

Client–Server

In a client–server (also called a point-to-point) communication protocol, components talk directly with other components. A good example of this is remote procedure call (RPC) protocols in which one component (the client) can call functions and procedures of another component (the server). A modern, and popular, variation on this is the common object request broker architecture (CORBA). CORBA allows for one component to call object methods that are implemented by another component. All method calls are defined in an interface definition language (IDL) file that is language independent. Every component uses the same IDL to generate code that compiles with component to handle communication. The advantage of this is that, when an IDL file is changed, all components that use that IDL can be recompiled automatically (by using *make* or similar code configuration tools). CORBA object request brokers (ORBs) are available for most major object-oriented languages. Although free ORBs are available, many commercial ORBs offer additional features and support. One disadvantage of CORBA is that it introduces quite a bit of additional code into applications. Some competitors have tried to address this issue, such as the internet communications engine (ICE), which has its own version of an IDL file called the specification language for ICE (SLICE). The biggest advantage of a client–server protocol is that the interfaces are very clearly defined in advance and everyone knows when the interface has changed. Another advantage is that it allows for a distributed approach to communication with no central module that must distribute data. A disadvantage of client–server protocols is that they introduce significant overhead, especially if many components need the same information. It should be noted that CORBA and ICE also have a broadcast mechanism (called an event channel, or the notification service, in CORBA).

Publish–Subscribe

In a publish–subscribe (also called a broadcast) protocol, a component publishes data and any other component can subscribe to that data. Typically, a centralized process routes data between publishers and subscribers. In a typical architecture, most components both publish information and subscribe to information published by other components. There are several existing publish–subscribe middleware solutions. A popular one for robotics is the real-time innovations' (RTI) data distribution service (DDS), formerly the network data distribution service (NDDS) [8.62]. Another popular publish–subscribe paradigm is IPC developed at Carnegie Mellon University [8.9]. Many publish-subscribe protocols are converging on using extensible markup language (XML) descriptions to define the data being published, with the added convenience of transmitting XML over HTTP, which allows for significant interoperability with Web-based applications. Publish–subscribe protocols have a large advantage in being simple to use and having low overhead. They are especially useful when it is unknown how many different components might need a piece of data (e.g., multiple user interfaces). Also, components do not get bogged down with repeated requests for information from many different sources. Publish–subscribe protocols are often more difficult to debug because the syntax of the message is often hidden in a simple string type. Thus problems are not revealed until runtime when a component tries, and fails, to parse an incoming message. Publish–subscribe protocols are also not as readable when it comes to sending commands from one module to another. Instead of calling an explicit method or function with parameters, a command is issued by publishing a message with the command and parameters in it and then having that message be parsed by a subscriber. Finally, publish–subscribe protocols often use a single central server to dispatch messages to all subscribers, providing a single point of failure and potential bottleneck.

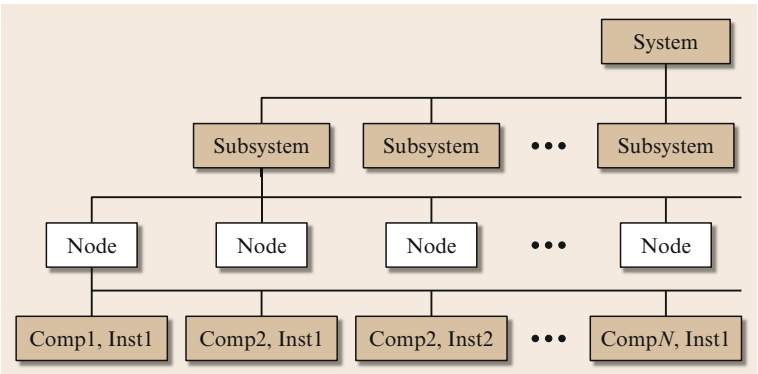

Fig. 8.7 The JAUS reference architecture topology (after JAUS Reference Architecture document [8.63])

JAUS

Recently, a standard has emerged in the defense robotics community not only for a communication protocol but also for definitions of messages that are sent via that communication protocol. The joint architecture for unmanned systems (JAUS) defines a set of reusable messages and interfaces that can be used to command autonomous systems [8.63–65]. These reusable components reduce the cost of integrating new hardware components into autonomous systems. Reuse also allows for components developed for one autonomous system to be used by another autonomous system. JAUS has two components: a *domain model* and a *reference architecture*. The domain model is a representation of the unmanned systems' functions and information. It contains a description of the system's functional and informational capabilities. The former includes models of the system's maneuvering, navigational, sensing, payload, and manipulation capabilities. The latter includes models of the system's internal data, such as maps and system status. The reference architecture provides a well-defined set of messages. Messages cause actions to commence, information to be exchanged, and events to occur. Everything that occurs in a JAUS system is precipitated by messages. This strategy makes JAUS a component-based, message-passing architecture.

The JAUS reference architecture defines a system hierarchy, as shown in Fig. 8.7. The topology defines the *system* as the collection of vehicles, operator control units (OCU), and infrastructure necessary to provide the full robotic capability. Subsystems are individual units (e.g., vehicles or OCUs) in the system. Nodes define a distinct processing capability within the architecture and route JAUS messages to components. Components provide the different execution capabilities and respond directly to command messages. Components might be sensors (e.g., a SICK laser or a vision sensor), actuators (a manipulator or a mobile base) or payloads (weapons or task sensors). The topology (the layout of particular system, subsystems, nodes, and components) is defined by the system implementers based on task requirements.

At the core of JAUS is a set of well-defined messages. JAUS supports the following message types.

Command:	Initiate mode changes or actions
Query:	Used to solicit information from a component
Inform:	Response to a query
Event set up:	Passes parameters to set up an event
Event notification:	Sent when the event happens

JAUS has about 30 predefined messages that can be used to control robots. There are messages for control of a robotic vehicle. For example, the *global vector driver* message performs closed-loop control of the desired global heading, altitude, and speed of a mobile vehicle. There are also sensor messages such as *global pose sensor*, which distributes the global position and orientation of the vehicle. There are also manipulation messages in JAUS. For example, the *set joint positions* message sets the desired joint position values. The *set tool point* message specifies the coordinates of the end-effector tool point in terms of the coordinate system attached to the end-effector.

JAUS also has user-definable messages. Messages have headers that follow a specific format and include message type, destination address (e.g., system, subsystem, node, and component), priority, etc. While JAUS is primarily point to point, JAUS messages can also be marked as *broadcast* and distributed to all components. JAUS also defines coordinate systems for navigation and manipulation to ensure all components understand any coordinates sent to them.

8.3.2 Behavioral Control

Behavioral control represents the lowest level of control in a robot architecture. It directly connects sensors and actuators. While these are typically hand-crafted functions written in C or C++, there have been specialized languages developed for behavioral control, including ALFA [8.66], Behavioral Language [8.67], and Rex [8.68]. It is at this level that traditional control theory (e.g., PID functions, Kalman filters, etc.) resides. In architectures such as 3T, the behavioral layer functions as a *Brooksian machine* – that is, the layer is composed of a small number of behaviors (also called skills) that perceive the environment and carry out the actions of the robot.

Example

Consider an office delivery robot that operates in a typical office building. The behavioral control layer contains the control functions necessary to move around in the building and carry out delivery tasks. Assuming the robot has an a priori map of the building, some possible behaviors for this robot include

1. move to location while avoiding obstacles
2. move down hallway while avoiding obstacles
3. find a door
4. find a door knob
5. grasp a door knob
6. turn a door knob
7. go through door
8. determine location
9. find office number
10. announce delivery

Each of these behaviors ties sensors (vision, range sensing, etc.) to actuators (wheel motors, manipulator motors, etc.) in a tight loop. In architectures such as Subsumption, all behaviors are running concurrently with a hierarchical control scheme inhibiting the outputs of certain behaviors. In AuRA [8.29], behaviors are combined using potential functions. Other architectures [8.24, 68] use explicit *arbitration* mechanisms to choose amongst potentially conflicting behaviors.

In architectures such as 3T [8.36], not all of the behaviors are active at the same time. Typically, only a few behaviors that do not conflict would be active at a time (e.g., behaviors 2 and 9 in the example above). The executive layer (see Sect. 8.3.3) is responsible for activating and deactivating behaviors to achieve higher-level tasks and to avoid conflicts between two behaviors competing for the same resource (e.g., an actuator).

Situated Behaviors

An important aspect of these behaviors is that they be *situated*. This means that the behavior works only in very specific situations. For example, behavior 2 above moves down a hallway, but this is appropriate only when the robot is situated in a hallway. Similarly, behavior 5, which grasps a door knob, is appropriate only when the robot is within grasping distance of a door knob. The behavior is not responsible for putting the robot in the particular situation. However, it should recognize that the situation is not appropriate and signal as such.

Cognizant Failure

A key requirement for behaviors is that they know when they are not working. This is called *cognizant failure* [8.69]. For example, behavior 5 in our example (grasping the door knob) should not continually grasp at air if it is failing. More succinctly, the behavior should not continue to *bang its head against the wall*. A common problem with early Subsumption robots is that the behaviors did not know they were failing and continued to take actions that were not resulting in progress. It is not the job of the behavioral control layer to decide what to do in a failure situation; it is only necessary to announce that the behavior has failed and halt activity.

Implementation Constraints

The behavioral control layer is designed to bring the speed and reactivity of Subsumption to robot control. For this reason, the behaviors in the behavioral control layer need to follow the philosophies of Subsumption. In particular, the algorithms used for behaviors should be constant in state and time complexity. There should be little or no search at the behavioral control level, and little iteration. Behaviors should simply be transfer functions that take in signals (from sensors or other behaviors) and send out signals (to actuators or other behaviors), and repeat these several times a second. This will allow for reactivity to changing environments. More controversial is how much state should be allowed at the behavioral level. *Brooks* famously said several years ago to "use the world as its own best model" [8.67] – that is, instead of maintaining internal models of the world and querying those models, the robot should instead directly sense the world to get its data. State such as maps, models, etc. belong at the higher levels of the three-tiered prototype architecture, not at the behavioral control layer. Certain exceptions, such as maintaining state for data filtering calculations, could be made on a case-by-case basis. *Gat* [8.70] argues that any state kept at the behavioral layer should be ephemeral and limited.

8.3.3 Executive

The executive layer is the interface between the numerical behavioral control and the symbolic planning layers. The executive is responsible for translating high-level plans into low-level behaviors, invoking behaviors at the appropriate times, monitoring execution, and handling exceptions. Some executives also allocate and monitor resource usage, although that functionality is more commonly performed by the planning layer.

Example

Continuing the example of an office delivery robot, the main high-level task would be to deliver mail to a given office. The executive would decompose this task into a set of subtasks. It may use a geometric path planner to determine the sequence of corridors to move down and intersections at which to turn. If there are doorways along the route, a task would be inserted to open and pass through the door. At the last corridor, the executive would add a concurrent task that looks for the office number. The final subtasks would be to announce that the person has mail and to concurrently monitor whether the mail has been picked up. If it is not picked up after some period of time, an exception would be triggered that invokes some recovery action (perhaps announcing again, perhaps checking to make sure the robot is at the correct office, perhaps notifying the planning layer to reschedule the delivery for a later time).

Capabilities

The example above illustrates many of the capabilities of the executive layer. First, the executive decomposes high-level tasks (goals) into low-level tasks (behaviors). This is typically done in a *procedural* fashion: the knowledge encoded in the executive describes *how* to achieve tasks, rather than describing *what* needs to be done and having the executive figure out the *how* by itself. Sometimes, though, the executive may also use specialized planning techniques, such as the route planner used in the example above. The decomposition is typically a hierarchical *task tree* (Fig. 8.8), with the leaves of the task tree being invocations and parameterizations of behaviors.

Besides decomposing tasks into subtasks, executives add and maintain temporal constraints between tasks (usually between sibling tasks only, but some executive languages permit temporal constraints between any pair of tasks). The most common constraints are *serial* and *concurrent*, but most executives support more expressive constraint languages, such as having one task begin 10 s after another one starts or having one task end when another ends.

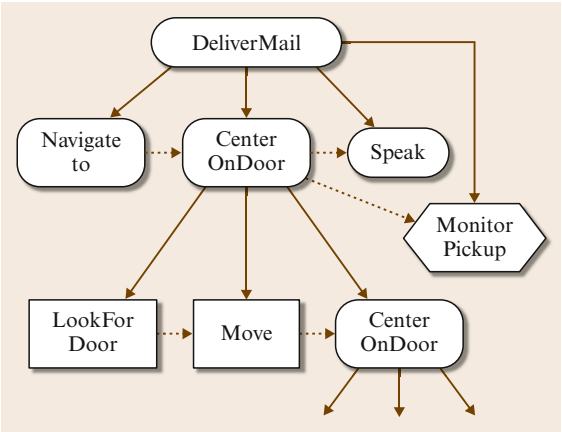

Fig. 8.8 Hierarchical task tree for mail-delivery task (*lozenge nodes* are interior; *rectangular nodes* are leaves; *hexagonal node* is an execution monitor; *solid arrows* are parent–child relationships; *dashed arrows* are sequential constraints)

The executive is responsible for dispatching tasks when their temporal constraints are satisfied. In some executives, tasks may also specify resources (e.g., the robot's motors or camera) that must be available before the task can be dispatched. As with behaviors, arbitrating between conflicting tasks can be a problem. In the case of executives, however, this arbitration is typically either programmed in explicitly (e.g., a rule that says what to do in cases where the robot's attempt to avoid obstacles takes it off the preferred route) or handled using priorities (e.g., recharging is more important than mail delivery).

The final two important executive capabilities are execution monitoring and error recovery. One may wonder why these capabilities are needed if the underlying behaviors are reliable. There are two reasons. First, as described in Sect. 8.3.2, the behaviors are situated, and the situation may change unexpectedly. For instance, a behavior may be implemented assuming that a person is available to pick up the mail, but that may not always be the case. Second, in trying to achieve some goal, the behavior may move the robot into a state that is unexpected by the executive. For instance, people may take advantage of the robot's obstacle avoidance behavior to *herd* it into a closet. While the behavior layer may, in fact, keep the robot safe in such situations, the executive needs to detect the situation in order to get the robot back on track.

Typically, execution monitoring is implemented as a concurrent task that either analyzes sensor data directly or activates a behavior that sends a signal to the executive when the monitored situation arises. These correspond to *polling* and *interrupt-driven* monitors, respectively.

Executives support various responses to monitors being triggered. A monitor may spawn subtasks that handle the situation, it may terminate already spawned subtasks, it may cause the parent task to fail, or it may raise an exception. The latter two responses involve the *error recovery* (also called *exception handling*) capability. Many executives have tasks return status values (success or failure) and allow parent tasks to execute conditionally based on the return values. Other executives use a hierarchical exception mechanism that throws named exceptions to ancestor nodes in the task tree. The closest task that has registered a handler for that exception tries to handle it; if it cannot, it rethrows the exception up the tree. This mechanism, which is inspired by the exception handling mechanisms of C++, Java, and Lisp, is strictly more expressive than the return-value mechanism, but it is also much more difficult to design systems using that approach, due to the nonlocal nature of the control flow.

Implementation Constraints

The underlying formalism for most executives is a hierarchical finite-state controller. Petri nets [8.71] are a popular choice for representing executive functions. In addition, various languages have been developed specifically to assist programmers in implementing executive-level capabilities. We briefly discuss aspects of several of these languages: reactive action packages (RAPs) [8.4, 30], the procedural reasoning system (PRS) [8.42, 43], the execution support language (ESL) [8.12], the task description language (TDL) [8.55], and the plan execution interchange language (PLEXIL) [8.13].

These languages all share features and exhibit differences. One distinction is whether the language is stand-alone (RAPs, PRS, PLEXIL) or an extension of an existing language (ESL is an extension of Common Lisp; TDL is an extension of C++). Stand-alone languages are typically easier to analyze and verify, but extensions are more flexible, especially with respect to integration with legacy software. While stand-alone executive languages all support interfaces to user-defined functions. These interfaces are usually limited in capability (such as what types of data structures can be passed around).

All of these executive languages provide support for hierarchical decomposition of tasks into subtasks. All except PLEXIL allow for recursive invocation of tasks. RAPs, TDL, and PLEXIL have syntax that distinguishes leaf nodes of the task tree/graph from interior nodes.

All these languages provide capabilities for expressing conditionals and iteration, although with RAPs and PLEXIL these are not core-language constructs, but must be expressed as combinations of other constructs. Except for TDL, the languages all provide explicit support for encoding pre- and post-conditions of the tasks and for specifying success criteria. With TDL, these concepts must be programmed in, using more primitive constructs. The stand-alone languages all enable local variables to be defined within a task description, but provide for only limited computation with those variables. Obviously, with extension languages the full capability of the base language is available for defining tasks.

All the languages support the simple serial (sequential) and concurrent (parallel) temporal constraints between tasks, as well as timeouts that can be specified to trigger after waiting a specified amount of time. In addition, TDL directly supports a wide range of temporal constraints – one can specify constraints between the start and end times of tasks (e.g., *task B starts after task A starts* or *task C ends after task D starts*) as well as metric constraints (e.g., *task B starts 10 seconds after task A ends* or *task C starts at 1pm*). ESL and PLEXIL support the signaling of events (e.g., when tasks transition to new states) that can be used to implement similarly expressive types of constraints. In addition, ESL and TDL support task termination based on the occurrence of events (e.g., *task B terminates when task A starts*).

The languages presented differ considerably in how they deal with execution monitoring and exception handling. ESL and TDL both provide explicit execution monitoring constructs and support exceptions that are *thrown* and then *caught* by registered handlers in a hierarchical fashion. This type of exception handling is similar to that used in C++, Java, and Lisp. ESL and TDL also support *clean-up* procedures that can be invoked when tasks are terminated. RAPs and PLEXIL use return values to signal failure, and do not have hierarchical exception handling. PLEXIL, though, does support clean up procedures that are run when tasks fail. PRS has support for execution monitoring, but not exception handling. ESL and PRS support the notion of *resources* that can be shared. Both provide support for automatically preventing contention amongst tasks for the resources. In the other executive languages, this must be implemented separately (although there are plans to extend PLEXIL in this area).

Finally, RAPs, PRS and ESL all include a symbolic database (*world model*) that connects either directly to sensors or to the behavior layer to maintain synchrony with the real world. Queries to the database are used to determine the truth of preconditions, to determine which methods are applicable, etc. PLEXIL has the concept of a *lookup* that performs a similar function, although it is transparent to the task how this is implemented (e.g., by a database lookup or by invoking a behavior-level function). TDL leaves it up to the programmer to specify how the tasks connect to the world.

8.3.4 Planning

The planning component of our prototype layered architecture is responsible for determining the long-range activities of the robot based on high-level goals. Where the behavioral control component is concerned with the here-and-now and the executive is concerned with what has just happened and what should happen next, the planning component looks towards the future. In our running example of an office delivery robot, the planning component would look at the day's deliveries, the resources of the robot, and a map, and determine the optimal delivery routes and schedule, including when the robot should recharge. The planning component is also responsible for replanning when the situation changes. For example, if an office is locked, the planning component would determine a new delivery schedule that puts that office's delivery later in the day.

Types of Planning

Chapter 9 describes approaches to robot planning in detail. Here, we summarize issues with respect to different types of planners as they relate to layered architectures.

The two most common approaches used are hierarchical task net (HTN) planners and planner/schedulers. HTN planners [8.72, 73] decompose tasks into subtasks, in a manner similar to what many executives do. The main differences are that HTN planners typically operate at higher levels of abstraction, take resource utilization into account, and have methods for dealing with conflicts between tasks (e.g., tasks needing the same resources, or one task negating a precondition needed by another task). The knowledge needed by HTN planners is typically fairly easy to specify, since one indicates directly *how* tasks are to be achieved.

Planner/schedulers [8.74, 75] are useful in domains where time and resources are limited. They create high-level plans that schedule when tasks should occur, but typically leave it to the executive to determine exactly how to achieve the tasks. Planner/schedulers typically work by laying out tasks on time lines, with separate time lines for the various resources that are available on the robot (motors, power, communication, etc.). The knowledge needed by planner/schedulers includes the goals that tasks achieve, the resources they need, their duration, and any constraints between tasks.

Many architectures provide for specialized planning *experts* that are capable of solving particular problems efficiently. In particular, these include motion planners, such as path planners and trajectory planners. Sometimes, the planning layer of the architecture invokes these specialized planners directly; in other architectural styles, the motion planners are part of the lower levels of the architecture (the executive, or even the behavioral layer). Where to put these specialized planners is often a question of style and performance (see Sect. 8.5).

Additionally, some architectures provide for multiple planning layers [8.39, 44, 76]. Often, there is a *mission* planning layer at the very top that plans at a very abstract level, over relatively long periods of time. This layer is responsible mainly for selecting which high-level goals are to be achieved over the next period of time (and, in some cases, determining in which order to achieve them) in order to maximize some objective function (e.g., net reward). The lower *task* planning layer is then responsible for determining exactly how and when to achieve each goal. This breakdown is usually done for efficiency reasons, since it is difficult to plan simultaneously at both a detailed level and over a long time horizon.

Integrating Planning and Execution

There are two main approaches to the integration of the planning and execution components in robotic architectures. The first approach is that the planning component is invoked as needed by the executive and returns a plan. The planning component is then dormant until called again. Architectures such as ATLANTIS [8.70] and Remote Agent [8.44] use this approach, which requires that the executive either leave enough time for planning to complete or that it *safes* the system until planning is complete. In the Remote Agent, for instance, a special *planning* task is explicitly scheduled.

The second approach is that the planning component sends high-level tasks down to the executive as required and monitors the progress of those tasks. If tasks fail, replanning is done immediately. In this approach, the planning component is always running and always planning and replanning. Signals must pass in real time between the planner and the executive to keep

them synchronized. Architectures such as 3T [8.36] use this second approach. The first approach is useful when the system is relatively static, so that planning can occur infrequently, at relatively predictable times. The second approach is more suited to dynamic environments, where replanning is more frequent and less predictable.

Other decisions that need to be made when integrating planning and execution are when to stop task decomposition, where to monitor plan execution, and how to handle exceptions. By planning all the way down to primitive actions/behaviors, the planner has a very good notion of what will happen during execution, but at a price of much more computation. Also, some task decompositions are easier to describe procedurally (using an executive language) rather than declaratively (using a planning language). Similarly, monitoring at the executive level tends to be more efficient, since the monitoring happens closer to the robot sensors, but the planner may be able to use its more global knowledge to detect exceptions earlier and/or more accurately. With respect to handling exceptions, executives can handle many on their own, at the price of breaking the expectations used by the planner in scheduling tasks. On the other hand, having exceptions handled by the planner typically involves replanning, which can be computationally expensive.

For all these integration issues, however, a middle ground usually exists. For instance, one can choose to decompose some tasks more deeply than others, or handle certain exceptions in the executive and others in the planner. In general, the *right* approach usually involves a compromise and is determined by analyzing the domain in detail (see Sect. 8.5).

8.4 Case Study – GRACE

In this section, we present the architecture of a fairly complex autonomous mobile robot. Graduate robot attending conference (GRACE) resulted from the efforts of five research institutions (Carnegie Mellon, Naval Research Laboratory, Northwestern University, Metrica, and Swarthmore College) to tackle the American Association for Artificial Intelligence (AAAI) Robot Challenge. The Challenge was for a robot to attend the AAAI National Conference on Artificial Intelligence as a participant – the robot must find the registration desk (without knowing the layout of the convention center beforehand), register for the conference, and then, after being provided with a map, find its way to a given location in time to give a technical talk about itself.

The architectural design of the robot was particularly important given the complexity of the task and the need to integrate techniques that had been previously developed by the five institutions. These techniques included localization in a dynamic environment, safe navigation in the presence of moving people, path planning, dynamic replanning, visual tracking of people, signs and landmarks, gesture and face recognition, speech recognition and natural language understanding, speech generation, knowledge representation, and social interaction with people.

GRACE is built on top of an real world interface (RWI) B21 base and has an expressive computer-animated face projected on a flat-panel liquid-crystal display (LCD) screen (Fig. 8.9). Sensors that come with the B21 include touch, infrared, and sonar sensors. Near the base is a SICK scanning laser range finder that provides a 180° field of view. In addition, GRACE has several cameras, including a stereo camera head on a pan–tilt unit (PTU) built by Metrica TRACLabs and a single-color camera with pan–tilt–zoom capability, built by Canon. GRACE can speak using a high-quality speech-generation software (Festival), and receive speech responses using a wireless microphone headset (a Shure TC computer wireless transmitter/receiver pair).

The behavioral layer of the architecture consisted of individual processes that controlled particular pieces of hardware. These programs provided abstract interfaces to either control the hardware or return information from

Fig. 8.9 The robot GRACE

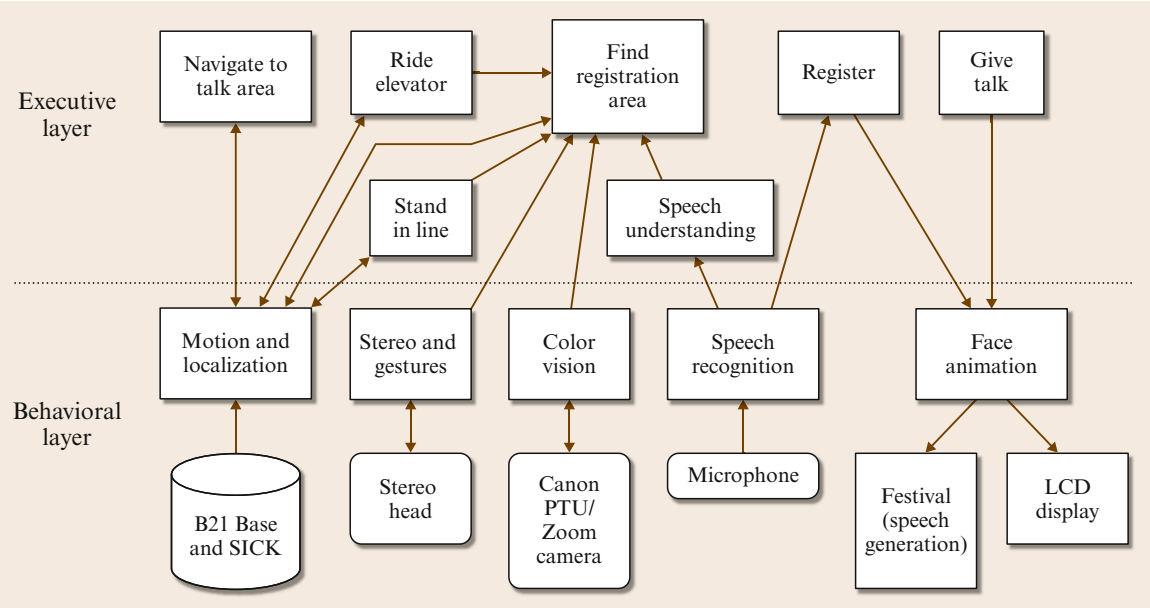

Fig. 8.10 GRACE's architectural structure

sensors. To accommodate the different coding styles of the various groups involved, both synchronous, blocking and asynchronous, nonblocking calls were supported by most of the interfaces (for the nonblocking calls, the interfaces allowed programmers to specify a callback function to be invoked when data was returned). Interfaces at the behavioral level included robot motion and localization (this interface also provided laser information), speech recognition, speech generation, facial animation, color vision, and stereo vision (Fig. 8.10).

The architecture used individual processes for each of the behavioral capabilities, mainly because the underlying code had been developed by different organizations. While having a large number of processes run concurrently is somewhat inefficient, trying to integrate everything into a monolithic process was thought to be too difficult. In addition, the use of separate processes facilitated development and debugging, since one needed to run only those aspects of the system that were being tested.

The executive layer consisted of separate programs for achieving each subtask of the challenge – finding the registration desk, riding the elevator, standing in line, interacting with the person at the desk, navigating to the talk, and giving the talk (Fig. 8.10). As is common in many implemented robotic systems, the GRACE architecture did not have a planning layer – since the high-level plan was fixed and relatively straightforward, it was coded explicitly. Several of the executive-layer programs were written using TDL (see Sect. 8.3.3),

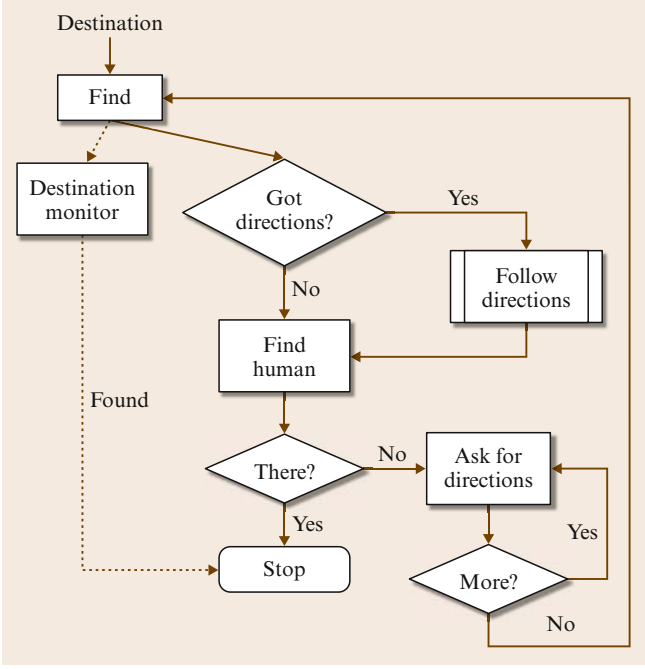

Fig. 8.11 Finite-state machine for GRACE's task for following directions to the registration booth

which facilitated concurrent control and monitoring of the various tasks.

One particularly involved task was finding the registration desk (recall that GRACE had no idea where the booth was, or even what the convention center looked like). TDL was used to create a finite-state machine that allowed GRACE to maintain multiple goals, such as using an elevator to get to a particular floor and following directions to find the elevator (Fig. 8.11). The top-level goal was to find the registration desk. Intermediate subgoals were created as GRACE interacted with people to determine the directions to the desk. If there were no directions to follow, GRACE performed a random walk until a person was detected using its laser scanner. GRACE then engaged in conversation with the person to obtain directions. GRACE could handle simple commands, such as *turn left* and *go forward five meters*, as well as higher-level instructions, such as *take the elevator* and *turn left at the next intersection*. In addition, GRACE could ask questions, such as *am I at the registration desk?* and *is this the elevator?* The TDL-based finite-state machine was used to determine which interactions were appropriate at various times and to prevent the robot from getting confused.

Communication between processes used the interprocess communication (IPC) messaging package [8.9, 77]. IPC supports both publish–subscribe and client–server messaging, and enables complex data structures to be passed transparently between processes. One side benefit of using IPC to communicate between processes was the ability to log all message traffic (both message name and data content). This proved invaluable, at times, in determining why the system failed to act as expected – did a process send out a message with invalid data? Did it fail to send out a message in a timely fashion? Was the receiving process blocked, for some reason? Was there a timing issue? While wading through the message traffic was often tedious, in some cases it was the only way to catch intermittent bugs.

In July 2002, GRACE successfully completed the challenge at the Shaw Convention Centre in Edmonton, Canada. The processes at the behavioral level generally worked as anticipated – this was largely attributed to the fact that those modules were ported from previously developed (and hence well-tested) systems. While generally functional, the executive-level processes had more problems with off-nominal situations. This is largely attributed to problems in sensor interpretation, as well as mistaken assumptions about what the convention center was going to look like (for instance, it turned out that some of the barriers were made of glass, which is largely invisible to the laser). Overall, however, the architecture itself worked as expected, enabling a large body of complex software to be integrated rather quickly and operate together effectively.

8.5 The Art of Robot Architectures

Designing a robot architecture is much more of an art than a science. The goal of an architecture is to make programming a robot easier, safer, and more flexible. Thus, the decisions made by a developer of a robot architecture are influenced by their own prior experiences (e.g., what programming languages they are familiar with), their robot and its environment, and the tasks that need to be performed. The choice of a robot architecture should not be taken lightly, as it is the authors' experiences that early architectural decisions often persist for years. Changing robot architectures is a difficult proposition and can set back progress while a great deal of code is reimplemented.

The art of designing a robotic architecture starts with a set of questions that the designer needs to ask. These questions include:

- What are the tasks the robot will be performing? Are they long-term tasks? Short-term? User-initiated? Robot-initiated? Are the tasks repetitive or different across time?
- What actions are necessary to perform the tasks? How are those actions represented? How are those actions coordinated? How fast do actions need to be selected/changed? At what speed do each of the actions need to run in order to keep the robot safe?
- What data is necessary to do the tasks? How will the robot obtain that data from the environment or from the users? What sensors will produce the data? What representations will be used for the data? What processes will abstract the sensory data into representations internal to the architecture? How often does the data need to be updated? How often can it be updated?
- What computational capabilities will the robot have? What data will these computational capabilities produce? What data will they consume? How will

the computational capabilities of a robot be divided, structured, and interconnected? What is the best decomposition/granularity of computational capabilities? How much does each computational capability have to know about the other capabilities? Are there legacy computational capabilities (from other robots, other robot projects, etc.) that will be used? Where will the different computational capabilities reside (e.g., onboard or offboard)?
- Who are the robot's users? What will they command the robot to do? What information will they want to see from the robot? What understanding do they need of the robot's computational capabilities? How will the user know what the robot is doing? Is the user interaction peer to peer, supervisory, or as a bystander?
- How will the robot be evaluated? What are the success criteria? What are the failure modes? What is the mitigation for those failure modes?
- Will the robot architecture be used for more than one set of tasks? For more than one kind of robot? By more than one team of developers?

Once designers have answers to all (or most) of these questions, they can then begin building some *use cases* for the types of operations they want the robot to perform and how they want users to interact with it. These use cases should specify the outward behavior of the robot with respect to its environment and its users. From the use cases, an initial partitioning of robot functionality can be developed. This partitioning should be accompanied by a *sequence diagram* that shows the transfer of information and control over time amongst the various components of the robot architecture [8.78]. After this, a more formal specification of the interfaces between architectural components can be developed. This may be done using a language such as the interface definition language (IDL) of CORBA or by defining the messages to be distributed in a publish–subscribe protocol. This is an important step, as once implementation begins it is very costly to change interfaces. If an interface does change, all stakeholders need to be notified and need to agree to the change. The most common integration problems in robot architectures are mismatches between what components expect and what they are receiving in the way of data.

An advantage of tiered architectures with clear interface definitions is that the different layers can be developed in parallel. The behavioral control layer can be implemented and tested on the robot using a human as an executive. The executive can be implemented and tested using state machine *stubs* for the expected behaviors on the robot. The planning layer can be implemented and tested using *stubs* for the tasks in the executive. The *stubs* merely acknowledge that they were called and report back appropriately. Then, the tiers can be integrated to test timing and other runtime issues. This parallel approach speeds up the development of a robot architecture, but is possible only if the roles and interfaces between components are clearly defined and respected. There is still considerable real-time debugging necessary during integration. In our experiences, most of the development time in robot architectures is still spent on the behavioral control layer – that is, sensing and acting are still the hard parts of robot control, as compared to execution and planning. Having a good, robust behavioral control layer goes a long way towards having a competent robot architecture.

8.6 Conclusions and Further Reading

Robot architectures are designed to facilitate the concurrent execution of task-achieving behaviors. They enable systems to control actuators, interpret sensors, plan, monitor execution, and deal with unexpected contingencies and opportunities. They provide the conceptual framework within which domain-dependent software development can take place, and they often provide programming tools that facilitate that development.

While no single architecture has proven to be best for all applications, researchers have developed a variety of approaches that can be applied in different situations. While there is not yet a specific formula for determining which architecture will be best suited for a given application, this chapter provides some guidelines to help developers in selecting the right architecture for the job. That being said, layered architectures have proven to be increasingly popular, due to their flexibility and ability to operate at multiple levels of abstraction simultaneously.

The book *AI and Mobile Robots* [8.79] has several chapters on architectures that have influenced this chapter. Most text books in robotics [8.19, 80, 81] have

sections on robot architectures. For many years in the mid 1990s, the AAAI Spring Symposia on Artificial Intelligence had sessions devoted to robot architectures, although proceedings from those symposia are not widely available. More information on GRACE can be found in [8.82–84].

References

8.1 I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process* (Addison Wesley Longman, Reading 1998)

8.2 J.S. Albus: RCS: A reference model architecture for intelligent systems, Working Notes: AAAI 1995 Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents (1995)

8.3 R.A. Brooks: A robust layered control system for a mobile robot, IEEE J. Robot. Autom. **2**(1), 14–23 (1986)

8.4 R.J. Firby: An Investigation into Reactive Planning in Complex Domains, Proc. of the Fifth National Conference on Artificial Intelligence (1987)

8.5 R. Simmons: Structured control for autonomous robots, IEEE Trans. Robot. Autom. **10**(1), 34–43 (1994)

8.6 J.J. Borrelly, E. Coste-Maniere, B. Espiau, K. Kapelos, R. Pissard-Gibollet, D. Simon, N. Turro: The ORCCAD architecture, Int. J. Robot. Res. **17**(4), 338–359 (1998)

8.7 B. Kuipers: The spatial semantic hierarchy, Artif. Intell. **119**, 191–233 (2000)

8.8 R. Orfali, D. Harkey: *Client/Server Programming with JAVA and CORBA* (Wiley, New York 1997)

8.9 R. Simmons, G. Whelan: Visualization Tools for Validating Software of Autonomous Spacecraft, Proc. of International Symposium on Artificial Intelligence, Robotics and Automation in Space (Tokyo 1997)

8.10 R. A. Brooks: The Behavior Language: User's Guide, Technical Report AIM-1227, MIT Artificial Intelligence Lab (1990)

8.11 R.J. Firby, M.G. Slack: Task execution: Interfacing to reactive skill networks, Working Notes: AAAI Spring Symposium on Lessons Learned from Implemented Architecture for Physical Agents (Stanford 1995)

8.12 E. Gat: ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents, Proc. of the IEEE Aerospace Conference (1997)

8.13 V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso: Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences, Proc. 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space (Munich 2005)

8.14 S.A. Schneider, V.W. Chen, G. Pardo-Castellote, H.H. Wang: ControlShell: A Software Architecture for Complex Electromechanical Systems, Int. J. Robot. Res. **17**(4), 360–380 (1998)

8.15 National Instruments: *LabVIEW* (National Instruments, Austin 2007), http://www.ni.com/labview/

8.16 N.J. Nilsson: A Mobile Automaton: An Application of AI Techniques, Proc. of the First International Joint Conference on Artificial Intelligence (Morgan Kaufmann Publishers, San Francisco 1969) pp. 509–520

8.17 N.J. Nilsson: *Principles of Artificial Intelligence* (Tioga, Palo Alto 1980)

8.18 P.E. Agre, D. Chapman: Pengi: An implementation of a theory of activity, Proc. of the Fifth National Conference on Artificial Intelligence (1987)

8.19 R.C. Arkin: *Behavior-Based Robotics* (MIT Press, Cambridge 1998)

8.20 J.H. Connell: SSS: A Hybrid Architecture Applied to Robot Navigation, Proc. IEEE International Conference on Robotics and Automation (1992) pp. 2719–2724

8.21 M. Mataric: Integration of Representation into Goal-Driven Behavior-Based Robots, Proc. IEEE International Conference on Robotics and Automation (1992)

8.22 I. Horswill: Polly: A Vision-Based Artificial Agent, Proc. of the National Conference on Artificial Intelligence (AAAI) (1993)

8.23 D.W. Payton: An Architecture for Reflexive Autonomous Vehicle Control, Proc. IEEE International Conference on Robotics and Automation (1986)

8.24 J.K. Rosenblatt: DAMN: A Distributed Architecture for Mobile Robot Navigation. Ph.D. Thesis (Carnegie Mellon Univ., Pittsburgh 1997)

8.25 R.C. Arkin: Motor schema-based mobile robot navigation, Int. J. Robot. Res. **8**(4), 92–112 (1989)

8.26 M. Arbib: Schema Theory. In: *Encyclopedia of Artificial Intelligence*, ed. by S. Shapiro (Wiley, New York 1992) pp. 1427–1443

8.27 O. Khatib: Real-time obstacle avoidance for manipulators and mobile robots, Proc. of the IEEE International Conference on Robotics and Automation (1985) pp. 500–505

8.28 R.C. Arkin: Integrating behavioral, perceptual, and world knowledge in reactive navigation, Robot. Autonom. Syst. **6**, 105–122 (1990)

8.29 R.C. Arkin, T. Balch: AuRA: Principles and practice in review, J. Exp. Theor. Artif. Intell. **9**(2/3), 175–188 (1997)

8.30 R.J. Firby: Adaptive Execution in Complex Dynamic Worlds. Ph.D. Thesis (Yale Univ., New Haven 1989)

8.31 R.J. Firby: Task Networks for Controlling Continuous Processes, Proc. of the Second International Conference on AI Planning Systems (1994)

8.32 R.P. Bonasso: Integrating Reaction Plans and layered competences through synchronous control, Proc. International Joint Conferences on Artificial Intelligence (1991)

8.33 S.J. Rosenschein, L.P. Kaelbling: The synthesis of digital machines with provable epistemic properties, Proc. of the Conference on Theoretical Aspects of Reasoning About Knowledge (1998)

8.34 L.P. Kaelbling: Goals as parallel program specifications, Proc. of the Sixth National Conference on Artificial Intelligence (1988)

8.35 L. P. Kaelbling: Compiling Operator Descriptions into Reactive Strategies Using Goal Regression, Technical Report, Teleos Research, TR90-10, (1990)

8.36 R.P. Bonasso, R.J. Firby, E. Gat, D. Kortenkamp, D.P. Miller, M.G. Slack: Experiences with an architecture for intelligent, reactive agents, J. Exp. Theor. Artif. Intell. **9**(2/3), 237–256 (1997)

8.37 E. Gat: Integrating Planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots, Proc. of the National Conference on Artificial Intelligence (AAAI) (1992)

8.38 G.N. Saridis: Architectures for Intelligent Controls. In: *Intelligent Control Systems: Theory and Applications*, ed. by Gupta, Sinhm (IEEE Press, Piscataway 1995)

8.39 R. Alami, R. Chatila, S. Fleury, M. Ghallab, F. Ingrand: An architecture for autonomy, Int. J. Robot. Res. **17**(4), 315–337 (1998)

8.40 M. Ghallab, H. Laruelle: Representation and control in IxTeT, a temporal planner, Proc. of AIPS-94 (1994)

8.41 P. Laborie, M. Ghallab: Planning with sharable resource constraints, Proc. of the International Joint Conference on Artificial Intelligence (1995)

8.42 M.P. Georgeff, F.F. Ingrand: Decision-Making in an Embedded Reasoning System, Proc. of International Joint Conference on Artificial Intelligence (1989) pp. 972–978

8.43 F. Ingrand, R. Chatila, R. Alami, F. Robert: PRS: A high level supervision and control language for autonomous mobile robots, Proc. of the IEEE International Conference On Robotics and Automation (1996)

8.44 N.P. Muscettola, P. Nayak, B. Pell, B.C. Williams: Remote agent: To boldly go where no AI system has gone before, Artif. Intell. **103**(1), 5–47 (1998)

8.45 B.C. Williams, P.P. Nayak: A Model-based Approach to Reactive Self-Configuring Systems, Proc. of AAAI (1996)

8.46 J.S. Albus: Outline for a theory of intelligence, IEEE Trans. Syst. Man Cybernet. **21**(3), 473–509 (1991)

8.47 B. Sellner, F.W. Heger, L.M. Hiatt, R. Simmons, S. Singh: Coordinated Multi-Agent Teams and Sliding Autonomy for Large-Scale Assembly, Proc IEEE **94**(7), 1425–1444 (2006), special issue on multi-agent systems

8.48 D. Goldberg, V. Cicirello, M.B. Dias, R. Simmons, S. Smith, A. Stentz: Market-Based Multi-Robot Planning in a Distributed Layered Architecture. In: *Multi-Robot Systems: From Swarms to Intelligent Automata*, Vol. II, ed. by A. Schultz, L. Parker, F.E. Schneider (Kluwer, Dordrecht 2003)

8.49 J.S. Albus, R. Lumia, H.G. McCain: NASA/NBS Standard Reference model for Telerobot Control System Architecture (NASREM), National Bureau of Standards, Tech Note #1235, NASA SS-GFSC-0027 (1986)

8.50 D.R. Blidberg, S.G. Chappell: Guidance and control architecture for the EAVE vehicle, IEEE J. Ocean Eng. **11**(4), 449–461 (1986)

8.51 R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, H. Das: The CLARAty architecture for robotic autonomy, Proc. of the IEEE Aerospace Conference (Big Sky 2001)

8.52 I.A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I. Shu, D. Apfelbaum: CLARAty: Challenges and steps toward reusable robotic software, Int. J. Adv. Robot. Syst. **3**(1), 023–030 (2006)

8.53 T. Estlin, D. Gaines, C. Chouinard, F. Fisher, R. Castaño, M. Judd, R. Anderson, I. Nesnas: Enabling Autonomous Rover Science Through Dynamic Planning and Scheduling, Proc. of IEEE Aerospace Conference (Big Sky 2005)

8.54 R. Knight, G. Rabideau, S. Chien, B. Engelhardt, R. Sherwood: CASPER: Space Exploration through Continuous Planning, IEEE Intell. Syst. **16**(5), 70–75 (2001)

8.55 R. Simmons, D. Apfelbaum: A Task Description Language for Robot Control, Proc. of Conference on Intelligent Robotics and Systems (Vancouver 1998)

8.56 T.A. Estlin, D. Gaines, C. Chouinard, R. Castaño, B. Bornstein, M. Judd, I.A.D. Nesnas, R. Anderson: Increased Mars Rover Autonomy using AI Planning, Scheduling and Execution, Proc. of the International Conference On Robotics and Automation (2007) pp. 4911–4918

8.57 D. Musliner, E. Durfee, K. Shin: World modeling for dynamic construction of real-time control plans, Artif. Intell. **74**(1), 83–127 (1995)

8.58 D.J. Musliner, R.P. Goldman, M.J. Pelican: Using Model Checking to Guarantee Safety in Automatically-Synthesized Real-Time Controllers, Proc. of International Conference on Robotics and Automation (2000)

8.59 B. Espiau, K. Kapellos, M. Jourdan: Formal Verification in Robotics: Why and How?, Proc. International Symposium on Robotics Research (Herrsching 1995)

8.60 G. Berry, G. Gonthier: The Esterel synchronous programming language: Design, semantics, implementation, Sci. Comput. Program. **19**(2), 87–152 (1992)

8.61 M. Jourdan, F. Maraninchi, A. Olivero: Verifying quantitative real-time properties of synchronous programs, Proc. 5th International Conference on Computer-aided Verification (Springer, Elounda 1993), LNCS 697

8.62 G. Pardo-Castellote, S.A. Schneider: The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications, Proc. of Inter-

national Conference on Robotics and Automation (1994) pp. 2870–2876
8.63 JAUS Reference Architecture Specification, Volume II, Part 1 Version 3.2 (available at http://www.jauswg.org/baseline/refarch.html)
8.64 JAUS Tutorial Powerpoint slides (available at: http://www.jauswg.org/)
8.65 JAUS Domain Model Volume I, Version 3.2 (available at http://www.jauswg.org/baseline/current_baseline.shtml)
8.66 E. Gat: ALFA: A Language for Programming Reactive Robotic Control Systems, Proc. IEEE International Conference on Robotics and Automation (1991) pp. 116–1121
8.67 R.A. Brooks: Elephants don't play chess, J. Robot. Autonom. Syst. **6**, 3–15 (1990)
8.68 L.P. Kaelbling: Rex- A symbolic language for the design and parallel implementation of embedded systems, Proc. of the 6th AIAA Computers in Aerospace Conference (Wakefield 1987)
8.69 E. Gat: Non-Linear Sequencing and Cognizant Failure, Proc. AIP Conference (1999)
8.70 E. Gat: On the role of stored internal state in the control of autonomous mobile robots, AI Mag. **14**(1), 64–73 (1993)
8.71 J.L. Peterson: *Petri Net Theory and the Modeling of Systems* (Prentice Hall, Upper Saddle River 1981)
8.72 K. Currie, A. Tate: O-Plan: The open planning architecture, Artif. Intell. **52**(1), 49–86 (1991)
8.73 D.S. Nau, Y. Cao, A. Lotem, H. Muñoz-Avila: SHOP: Simple hierarchical ordered planner, Proc. of the International Joint Conference on Artificial Intelligence (1999) pp. 968–973
8.74 S. Chien, R. Knight, A. Stechert, R. Sherwood, G. Rabideau: Using iterative repair to improve the responsiveness of planning and scheduling, Proc. of the International Conference on AI Planning and Scheduling (2000) pp. 300–307
8.75 N. Muscettola: HSTS: Integrating planning and scheduling. In: *Intelligent Scheduling*, ed. by M. Fox, M. Zweben (Morgan Kaufmann, San Francisco 1994)
8.76 R. Simmons, J. Fernandez, R. Goodwin, S. Koenig, J. O'Sullivan: Lessons Learned From Xavier, IEEE Robot. Autom. Mag. **7**(2), 33–39 (2000)
8.77 R. Simmons: *Inter Process Communication* (Carnegie Mellon Univ., Pittsburgh 2007), www.cs.cmu.edu/IPC
8.78 S.W. Ambler: *UML 2 Sequence Diagramms* (Ambisoft, Toronto 2007), www.agilemodeling.com/artifacts/sequenceDiagram.htm
8.79 D. Kortenkamp, R.P. Bonasso, R. Murphy: *Artificial Intelligence and Mobile Robots* (AAAI Press/The MIT Press, Cambridge 1998)
8.80 R. Murphy: *Introduction to AI Robotics* (MIT Press, Cambridge 2000)
8.81 R. Siegwart, I.R. Nourbakhsh: *Introduction to Autonomous Mobile Robots* (MIT Press, Cambridge 2004)
8.82 R. Simmons, D. Goldberg, A. Goode, M. Montemerlo, N. Roy, B. Sellner, C. Urmson, A. Schultz, M. Abramson, W. Adams, A. Atrash, M. Bugajska, M. Coblenz, M. MacMahon, D. Perzanowski, I. Horswill, R. Zubek, D. Kortenkamp, B. Wolfe, T. Milam, B. Maxwell: GRACE: An autonomous robot for the AAAI Robot Challenge, AAAI Mag. **24**(2), 51–72 (2003)
8.83 R. Gockley, R. Simmons, J. Wang, D. Busquets, C. DiSalvo, K. Caffrey, S. Rosenthal, J. Mink, S. Thomas, W. Adams, T. Lauducci, M. Bugajska, D. Perzanowski, A. Schultz: Grace and George: Social Robots at AAAI, AAAI 2004 Mobile Robot Competition Workshop (AAAI Press, 2004), Technical Report WS-04-11, pp. 15–20
8.84 M.P. Michalowski, S. Sabanovic, C. DiSalvo, D. Busquets, L.M. Hiatt, N.A. Melchior, R. Simmons: Socially Distributed Perception: GRACE plays social tag at AAAI 2005, Auton. Robot. **22**(4), 385–397 (2007)