On Three-Layer Architectures

Erann Gat
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109

(Appears in *Artificial Intelligence and Mobile Robots*, David Kortenkamp, R. Peter Bonnasso, and Robin Murphy, eds., AAAI Press)

1. Historical Background

In the mid-1980's Rodney Brooks touched off a firestorm of interest in autonomous robots with the introduction of the Subsumption architecture[1] [Brooks86]. At the time, the dominant view in the AI community was that a control system for an autonomous mobile robot should be decomposed into three functional elements: a sensing system, a planning system, and an execution system [Nilsson80]. The job of the sensing system is to translate raw sensor input (usually sonar or vision data) into a world model. The job of the planner is to take the world model and a goal and generate a plan to achieve the goal. The job of the execution system is to take the plan and generate the actions it prescribes.

The sense-plan-act (SPA) approach has two significant architectural features. First, the flow of control among these components is unidirectional and linear. Information flows from sensors to world model to plan to effectors, never in the reverse direction. Second, the execution of an SPA plan is analogous to the execution of a computer program. Both are built of primitives composed using partial orderings, conditionals, and loops. *Executing* a plan or a program is easy (and therefore uninteresting) when compared with *generating* one. The information content is in the composition structure, not the primitives. The intelligence of the system (such as it is) lives in the planner or the programmer, not the execution mechanism. Research efforts through about 1985 focused almost exclusively on planning and world modeling.

By 1985 it was becoming clear that SPA had numerous shortcomings. Planning and world modeling turned out to be Very Hard Problems, and open-loop plan execution was clearly inadequate in the face of environmental uncertainty and unpredictability. Several researchers in the mid 80's suggested that a different execution mechanism was needed [Firby87, Agre87, Payton86]. The SPA approach was so dominant at the time that this new work was labeled with the self-contradictory buzzword "reactive planning."

Subsumption is the best known departure from SPA. It is also popularly perceived as the most radical of its time. Ironically, in Brooks's seminal 1986 paper Subsumption is presented as a compatible extension to SPA:

> But what about each individual layer? Don't we need to decompose a single layer in the traditional manner? This is true to some extent, but the key difference is that we don't need to account for all desired perceptions and processing and generated behaviors in a single decomposition.

So in its original presentation, Subsumption is, at least ostensibly, not a radical departure from SPA at all, but rather an attempt to make SPA more efficient by applying task-dependent constraints to the Subsumption layers. This view is reinforced by the canonical diagram of Subsumption, which shows all information flowing unidirectionally from sensors to actuators, just as in SPA.

Where Subsumption departs radically from SPA is in its repudiation of plans (and indeed of symbolic representations in general [Brooks90]). From the details of Brooks's examples it is clear that Subsumption's layers are not decomposed "in the traditional manner" at all. Instead, they are simple networks of small finite state machines joined by "wires" which connect output ports to input ports. Subsumption provides only one mechanism for composing these basic building blocks to produce complex control mechanisms: the ability to override the contents of one wire with a value from another wire. This process is called suppression or inhibition depending on whether it takes place at an input or an output port. Subsumption also advocates a development methodology consisting of layering

[1] There is no consensus on the definition of the word "architecture"in the context of software systems. In this chapter I will use the word to mean a set of constraints on the structure of a software system.

progressively more complex task-specific control programs (called *behaviors*) on top of each other. However, Subsumption provides no architectural mechanism to enforce (or even support) this methodology.

Subsumption achieved dramatic early success in the area of collision-free robot navigation. While SPA-based robots were pondering their plans, Subsumption-based robots were zipping around the lab like R2D2. By the common metric that speed equals intelligence, Subsumption appeared to be a major breakthrough.

Subsumption reached a pinnacle with a robot called Herbert, which was programmed to find and retrieve soda cans in an office environment [Connell89]. (Brooks has recently launched an even more ambitious project called Cog, but as of this writing no results have been published.) While Herbert's capabilities were impressive even by today's standards, it also appeared to represent the limits of what could be achieved with Subsumption. Herbert was very unreliable (there is no record of it ever having performed a complete can-retrieval task flawlessly), and no Subsumption-based robot has ever matched its capabilities since.

One possible cause of Subsumption's apparent "capability ceiling" is that the architecture lacks mechanisms for managing complexity. Quoting from [Hartley91]:

> The most important problem we found with the Subsumption architecture is that is it not sufficiently modular. The other problems described below are really side-effects of this one. Because upper layers interfere with the internal functions of lower-level behaviors they cannot be designed independently and become increasingly complex. This also means that even small changes to low-level behaviors or to the vehicle itself cannot be made without redesigning the whole controller.
>
> Brooks proposes to solve this problem by reducing or even eliminating direct communications between modules. Instead behaviors would "communicate through the world." Except in a few cases ... we did not find this approach useful. The problem was that very similar states of the world could mean different things depending on the context. ...
>
> Determining that one behavior is more high-level than another is sometimes completely artificial. … Subsumption of low-level behaviors by high-level ones is not always appropriate. Sometimes the low level should override higher levels. ...

Note that Hartley is taking issue with the most fundamental tenet of Subsumption as a design methodology, saying in effect that the central (indeed the only) architectural mechanism that Subsumption provides often doesn't work. (It should be noted that Hartley's critique only addresses Subsumption as an engineering methodology, not as a model of human intelligence [Brooks91]. For such a critique, see [Kirsh91].)

The years following the introduction of Subsumption in 1986 saw a profusion of new robot control architectures, some developed more or less independently (e.g. [Kaelbling88], [Soldo90], [Arkin90], [Georgeff87], [Simmons90]) and others introduced as a direct response to Subsumption's shortcomings (e.g. [Rosenblatt89]). One of the first robots to be built using one of these latter alternatives was Tooth [Gat94] which was completed in the summer of 1989 [Angle89]. Tooth was a small robot (30 cm by 20 cm) with simple sensors and limited computation (two 8-bit microcontroller, each with about 2000 bytes of memory), but it was a very capable robot nonetheless. Tooth was programmed to search for small objects (styrofoam coffee cups), pick them up in its gripper, and return them to a light-bulb beacon. A similar capability was demonstrated a year later on an outdoor robot, RockyIII [Miller91, Gat94] using the same control methodology. In contrast with Herbert, Tooth and Rocky III were extremely reliable, running many dozens of trials without failing. (To be fair, Herbert was a much more ambitious robot, finding its soda cans using a structured-light vision system.)

The software that controlled Tooth and Rocky III (which I will refer to as T/R-III, not to be confused with the 3T architecture described later in this chapter) was a layered design like Subsumption. However, unlike Subsumption, T/R-III embraced abstraction rather than rejecting it. In Subsumption higher-level layers interface with lower level ones by *suppressing* the results of the lower-level computations and superseding their results. In T/R-III, higher-level layers interfaced with lower-level ones by *providing input or advice* to the lower-level layers (cf. [Payton90, Agre90]). In other words, layers in T/R-III provided layers of *computational abstraction* as well as layers of functionality.

Tooth and Rocky III were among the first autonomous robots capable of reliably performing a more complex task than simply moving from place to place, but they had one serious drawback: they were not *taskable*, that is, it was not possible to

change the task they performed without rewriting their control program.

At least three different groups of researchers working more or less independently came up with very similar solutions to this problem at about the same time [Connell91, Gat91, Bonasso91]. All three solutions consisted of control architectures that comprised three main components: a reactive feedback control mechanism, a slow deliberative planner, and a sequencing mechanism that connected the first two components. Connell's sequencer was based on Subsumption, Bonasso used Kaelbling's REX/GAPPS system [Kaelbling89], and Gat's was based on Firby's Reactive Action Packages (RAPs) system as described in his 1989 thesis [Firby89]. Bonasso's group later adopted RAPs as their sequencing component, while Gat's sequencer was recently developed into a new language, ESL [Gat97].

Aside from the technical advances, there are two items of historical interest in Firby's thesis. The first is that the title catch phrase was changed from "reactive planning" to "reactive execution," heralding a clean break from the SPA tradition. The second is that it contains the earliest description of the three-layer architecture that has now become the de facto standard [Firby89, figures 1.1 and 7.1]. This original three-layer architecture was briefly implemented on JPL's Robbie robot [Wilcox87], but there is no record of the results. RAPs has since been used to control a number of real robots, including Uncle Bob [Elsaessar&Slack94], Homer [Gat&Dorais94], and Chip [Firby96]. The RAP-based three-layer architecture has come be called 3T [Bonnasso et al. 96]. Connell's Subsumption-based architecture is called SSS. Gat's architecture is called ATLANTIS. It was first implemented on Robby in 1990 [Gat91,92], and has since been implemented on a number of other robots. (See section 4.) The main differences between 3T and ATLANTIS are that 1) ATLANTIS used a different representation in its sequencing layer, one designed more for programming convenience than for use as a planner representation, 2) the sequencer controlled the operation of the planner rather than vice versa. ATLANTIS also extended the then-existing RAPs action model to use continuos real-time processes rather than atomic operators, a feature which has since been incorporated back into the de facto standard.

2. The role of internal state

At this point the question naturally arises: why do so many independently designed architectures turn out to have such a similar structure? Are three components necessary and/or sufficient, or is three just an aesthetically pleasing number or a coincidence? I believe that there is a sound architectural rationale for having exactly three major components. It has to do with the role of internal state.

By way of motivation, consider the classic SPA architecture, and two of its associated difficulties. First, because planning is time-consuming, the world may change during the planning process in a way that invalidates the resulting plan. (The same problem exists for world modeling.) Second, an unexpected outcome from the execution of a plan step can cause subsequent plan steps to be executed in an inappropriate context. (This problem often manifests itself as "running researcher syndrome," characterized by having to chase the robot to push the emergency stop button after it makes a wrong turn.)

To be fair, let us also consider a problem associated with Brooks-style reactive architectures. A reactive robot using ultrasonic sensors to control its motions sometimes can collide with obstacles when specular (mirror-like) reflections produce readings that fail to indicate the obstacle's presence.

All three of these problems can be viewed as a result of the method used to manage stored internal state information [Gat93]. Time-consuming computations like planning and world modeling generate internal state whose semantics reflect world states, whether they are past, present (in the case of world models) or future (in the case of plans). Plan execution also involves internal state, the program counter, which implicitly encodes the planner's expectations about the state of the world as execution proceeds. SPA gets into trouble when its internal state loses sync with the reality that it is intended to represent.

The reactive solution to this problem is to minimize the use of internal state as much as possible. If there is no state, then it cannot lose sync with the world, a sentiment often expressed by the slogan, "The world is its own best model." Unfortunately, extracting information from the world-as-its-own-model requires using sensors, which are unreliable and subject to occlusions. Sometimes a robot might do well to remember that there was a wall in front of it a little while ago, and to conclude that the wall is probably still there despite the fact that it seems to have vanished according to the sonars. By eliminating internal state the reactive approach avoids the problem associated with maintaining that state, but runs headlong into the problem of extracting reliable information about the world through sensors.

Three-layer architectures organize algorithms according to whether they contain no state, contain state reflecting memories about the past, or contain state reflecting predictions about the future. Stateless sensor-based algorithms inhabit the control component. Algorithms that contain memory about the past inhabit the sequencer. Algorithms that make predictions about the future inhabit the deliberator.

Abstraction is used as a tool to isolate aspects of reality that can be tracked or predicted reliably, and ignore aspects that cannot.

3. The Anatomy of the Three Layer Architecture

The three-layer architecture consists of three components: a reactive feedback control mechanism, a reactive plan execution mechanism, and a mechanism for performing time-consuming deliberative computations. These components run as separate computational processes. This is most easily accomplished by using a multi-tasking or multi-threaded operating system, but can also be done my carefully coding the algorithms so they can be manually interleaved within a single computational process.

In 3T the components are called the *skill layer*, the *sequencing layer*, and the *planning layer* respectively. In ATLANTIS these layers are called the *controller*, the *sequencer*, and the *deliberator*. The following discussion uses the ATLANTIS terminology, but as much as possible the description is generic to all incarnations of the three-layer architecture.

3.1 The Controller

The controller consists of one or more threads of computation that implement one or more feedback control loops, tightly coupling sensors to actuators. The transfer function(s) computed by the controller can be changed at run time. Usually the controller contains a library of hand-crafted transfer functions (called primitive behaviors or skills). Which ones are active at any given time is determined by an external input to the controller.

To distinguish between the code that implements a transfer function, and the physical behavior produced by that transfer function when running on a robot in an environment, we shall capitalize the former. Thus, a Behavior is a piece of code that produces a behavior when it is running. Primitive Behaviors are designed to produce simple primitive behaviors that can be composed to produce more complex task-achieving behavior (a job done, naturally, by the sequencer). Classic examples of primitive behaviors are wall-following, moving to a destination while avoiding collisions, and moving through doorways.

There are several important architectural constraints on the algorithms that go into the controller. First, computing one iteration of the transfer function should be of constant-bounded time and space complexity, and this constant should be small enough to provide enough bandwidth to afford stable closed-loop control for the desired behavior.

Second, the algorithms in the controller should *fail cognizantly*, that is, they should be designed to detect (as opposed to avoid) any failure to perform the function for which they were designed (c.f. [Noreils90]). Rather than attempt to design algorithms that never fail (which is impossible on real robots) one can instead design algorithms that never fail to detect a failure. This allows other components of the system (the sequencer and deliberator) to take corrective action to recover from the failure.

Third, the use of internal state should be avoided whenever possible. An important exception to this rule is filtering algorithms, which rely on internal state, but can nevertheless be used in the controller. If internal state is used for other purposes, it should be *ephemeral*, that is, it should expire after some constant-bounded time. This way, if the semantics of the internal state do not reflect the true state of affairs in the environment at least the time during which this error will manifest itself will be bounded.

Finally, internal state in the controller should not introduce discontinuities (in the mathematical sense) in a Behavior. In other words, a Behavior (which is a transfer function) should be a continuous function with respect to its internal state. It is the responsibility of the sequencer to manage transitions between regimes of continuous operation.

A number of special-purpose languages have been developed for programming the controller (e.g. [Gat91b], [Brooks89]), but any language can be used as long as the architectural constraints are observed. Most of the special-purpose languages for programming the controller were developed at a time when robots could only support very small processors for which no other development tools were available. The current trend is to simply program the controller in C.

3.2 The Sequencer

The sequencer's job is to select which primitive Behavior (i.e. which transfer function) the controller should use at a given time, and to supply parameters for the Behavior. By changing primitive Behaviors at strategic moments the robot can be coaxed into performing useful tasks. The problem, of course, is that the outcome of selecting a particular primitive in a particular situation might not be the intended one, and so a simple linear sequence of primitives is unreliable. The sequencer must be able to respond conditionally to the current situation, whatever it might be.

One approach to the problem is to enumerate all the possible states the robot can be in, and precompute the correct primitive to use in each state for a

particular task. Clever encoding can actually make this daunting task tractable for certain constrained domains [Schoppers87]. However, this *universal plan* approach has two serious drawbacks. First, it is often not possible for a robot to know its current state, especially when unexpected contingencies arise. Second, this approach disregards the robot's execution history, which often contains useful information.

An alternative is to use an approach called *conditional sequencing*, which is a more complex model of plan execution motivated by human instruction following. Humans can achieve tasks based on very concise instructions in the face of a wide variety of contingencies (e.g. [Agre90], [Suchman87]). Conditional sequencing provides a computational framework for encoding the sort of procedural knowledge contained in instructions. It differs from traditional plan execution in that the control constructs for composing primitives are not limited to the partial ordering, conditionals, and loops used to build SPA plans. Conditional sequencing systems include constructs for responding to contingencies, and managing multiple parallel interacting tasks.

It is possible to construct a conditional sequencing system in a traditional programming language like C, but because the control constructs are so much more complex than those provided by such languages conditional sequencing is much more effectively done with a special-purpose language like RAPs [Firby89], PRS [Georgeff87], the Behavior Language [Brooks89], REX/GAPPS [Kaelbling87, Kaelbling89, Bonasso92], or ESL [Gat97].

There are two major approaches to the design of conditional sequencing languages. They can be complete languages in their own right with their own specialized execution semantics. RAPs and PRS take this approach. Or they can be layered on top of a syntactically extensible programming language like Lisp. This is the approach taken by the Behavior Language and ESL. Furthermore, the structure of the language can treat all possible outcomes of an action in a homogeneous fashion, or the language can be structured to recognize a privileged "nominal" result of an action and treat all other outcomes as "failures." Again, RAPs and PRS take the first approach; ESL takes the second.

Which approach one chooses depends on what one it trying to do. The RAPs/PRS approach results in a more circumscribed language that is suitable for use as a representation for an automated planner. The ESL approach, because it subsumes a traditional programming language, is more convenient to use and easier to extend, but more difficult to analyze.

The sequencer should not perform computations that take a long time relative to the rate of environmental change at the level of abstraction presented by the controller. Exactly how long a "long time" is depends on both the environment and the repertoire of Behaviors. Usually this constraint implies that the sequencer should not perform any search or temporal projection, but it might also constrain, for example, certain vision processing algorithms from appearing in the sequencer, especially if computational resources are limited.

3.3 The Deliberator

The deliberator is the locus of time-consuming computations. Usually this means such things as planning and other exponential search-based algorithms, but as noted before, it could also include polynomial-time algorithms with large constants such as certain vision processing algorithms in the face of limited computational resources. The key architectural feature of the deliberator is that several Behavior transitions can occur between the time a deliberative algorithm is invoked and the time it produces a result. The deliberator runs as one or more separate threads of control. There are no architectural constraints on algorithms in the deliberator, which are invariably written using standard programming languages.

The deliberator can interface to the rest of the system in two different ways. It can produce plans for the sequencer to execute, or it can respond to specific queries from the sequencer. The RAPs-based 3T architecture takes the first approach [Bonasso et al. 97]. The ESL-based ATLANTIS architecture takes the second approach. This is a natural result of the fact that RAPs was designed specifically to serve as a plan representation for an automated planning system and ESL was not. These two approaches are not mutually exclusive. RAPs does permit deliberative algorithms (called RAP-experts) to be invoked at runtime to answer specific queries, and the ATLANTIS sequencer can ask the deliberator to give it a complete plan which it then executes. (This is being done in an application of ESL to autonomous spacecraft [Pell et al. 96].)

3.4 Discussion

The architectural guidelines that govern the design of the three-layer architecture are not derived from fundamental theoretical considerations. Instead, they are derived from empirical observations of the properties of environments in which robots are expected to perform, and of the algorithms that have proven useful in controlling them. Robot algorithms tend to fall into three major equivalence classes: fast, mostly stateless reactive algorithms with hard real-time bounds on execution time, slow deliberative algorithms like planning, and intermediate algorithms

which are fairly fast, but cannot provide hard real-time guarantees.

"Fast" and "slow" are measured with respect to the rate of change of the environment. In principle, if the rate of change of the environment is sufficiently slow (or, equivalently, if a planner were available that was sufficiently fast) the controller could contain a planner. (Note that this situation is essentially equivalent to the SPA architecture.) Empirically it turns out not to be possible to build planners that are fast enough to operate in this manner in realistic environments.

4. Case Study

To date at least half a dozen different robots have been programmed using some variation of the three-layer architecture [Gat92, Gat&Dorais94, Nourbakhsh et al. 93, Elsaessar&Slack94, Connell91, Firby96, Bonasso et al. 97, Firby&Slack95. I will describe one of these here in some detail.

Alfred is a B12 robot built by Real World Interface (RWI). The B12 is a cylindrical robot, twelve inches in diameter with a synchrodrive mobility mechanism. Encoders on the drive and steering motors provide fairly reliable odometry and dead reckoning, although the robot's heading tends to precess due to slight misalignments of its wheels. A development enclosure houses a Gespak 68000 computer and a radially symmetric ring of twelve Polaroid sonars [Biber80]. The sonars are mounted on panels that allow them to be easily reconfigured. The sonar configuration was rotated 15 degrees from the default factory configuration, resulting in one sonar pointing straight forward in the direction of motion, and one sonar pointing directly to either side. (See figure 1.) This turned out empirically to make wall-following more reliable.

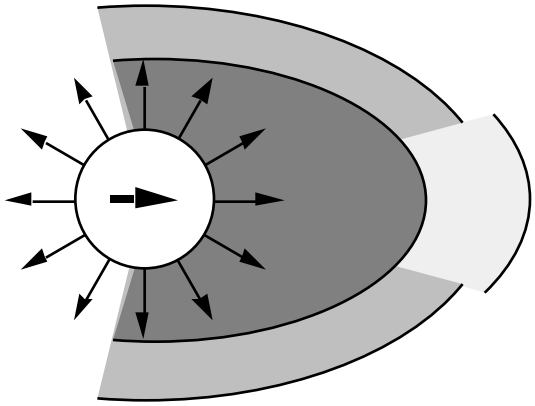

Figure 1: Schematic top-view of Alfred showing sonar directions (radial arrows) relative to the direction of motion (dark central arrow) and obstacle-detection regions.

Alfred also used a Macintosh Powerbook Duo230 running Macintosh Common Lisp (MCL) mounted on top of the robot. The Duo was connect to the Gespak board through an RS-232 serial port running at 9600 baud. Alfred's controller ran on the Gespak board. The sequencer and deliberator were programmed in Lisp and ran on the Powerbook.

Alfred was programmed to compete in two events at the 1993 AAAI mobile robot contest [Nourbakhsh at al. 93]. The first event was called "escape from the office" and involved finding a route out of a room filled with furniture, and across a large open area filled with boxes. The second event was called "deliver the coffee" and involved self-localization and path-planning in a maze.

Alfred placed second in the first event, and was the only robot to complete the second event. All the contest-specific code for the robot was written in three days by one person.

The following sections describe the control, sequencing, and deliberative layers on Alfred. These descriptions are faithful to the actual implementation used in the contest, and could no doubt be improved on.

4.1 Control Layer

Alfred's control layer was implemented in ALFA (A Language For Action) [Gat91b], a language designed by its structure to enforce the control layer's architectural constraints. ALFA is a dataflow language with no looping constructs. It does,

however, have state variables, making it Turing-complete. It is therefore possible to implement arbitrary algorithms in ALFA, and so ALFA's constraint enforcement is far from perfect. The language does not make it impossible to violate the rules, just more difficult. Unfortunately, it turns out that ALFA's design also makes it more difficult to do things that should be allowed in the control layer. I no longer advocate the use of ALFA, preferring instead to use C or Lisp and a little self-discipline to enforce the architectural constraints.

Alfred's control layer had four interesting behaviors: obstacle avoidance, wall-finding, wall-alignment, wall-following, and wandering. (It also had a number of uninteresting but nonetheless useful behaviors like turning in place.)

Obstacle avoidance was done as follows. First, the sonar data was preprocessed to indicate the presence or absence of an obstacle in each of five regions around the front of the robot. (The five rear-facing sonars were not used.) There were two near-field "hard obstacle" regions close to the front of the robot (figure 1, dark shading), and three "soft" obstacle regions further away. The hard obstacle region was divided into left and right regions that overlapped at the front sonar. The robot was able to detect collisions by monitoring its motor current. The obstacle regions were egg-shaped, extending further from the robot towards the front than at the sides.

At the core of the controller code was the following safety algorithm that was always running:

```
IF there is a collision while moving forward
  BACK UP slowly for a few seconds
ELSE IF there is a collision while moving
backwards
  STOP for a few seconds
ELSE IF there is an obstacle in one of the hard
obstacle regions
  STOP
ELSE IF there is an obstacle in one of the soft
obstacle regions
  set the current speed to SLOW
ELSE (there are no obstacles)
  gradually increase forward speed up to a
maximum value.
```

This code had the effect of slowing the robot down in the presence of obstacles, and stopping the robot when it was in imminent danger of collision. By allowing any detected obstacle to immediately slow the robot down, but only a succession of clear readings to speed it back up again, the robot reliably slowed down in cluttered areas even if when there was a lot of specular reflection.

Note that this code uses internal state to remember collisions for a few seconds after they happen, and to keep track of the current maximum speed. This use of internal state obeys the controller's architectural constraints because it is ephemeral, and in the second case it is part of a filtering algorithm. The filtering algorithm might appear to violate the prohibition on state-dependent discontinuities, but this is not the case. The output of the controller is a continuous function of the state; it is the value of the state that changes discontinuously over time. Ideally the collision response routine would have been put in the sequencer, but because the robot detected collisions by monitoring motor current, by the time a collision was detected there was already quite a bit of mechanical stress built up in the robot's drive mechanism. Simply stopping the robot would have caused the robot's motor servo controller to attempt to maintain the motor's velocity at zero, which would have maintained this mechanical stress. Relieving the stress required backing up, and to accomplish this as quickly as possible, the response was implemented in the controller. This is a good example of how the lines between the components of the three-layer architecture can be blurred to accommodate reality.

Obstacle avoidance was done with the following algorithm:

```
IF there is an obstacle in the soft-left
obstacle region and not in the soft-right
region
  turn slowly to the right
ELSE IF there is an obstacle in the soft-right
obstacle region but not in the soft-left
obstacle region
  turn slowly to the left
ELSE
  go straight, or turn towards a commanded
heading.
```

This algorithm only avoids obstacles when the choice of turning direction is clearly dictated by the situation. When an obstacle is directly in front, the robot does not turn. This is because the control layer has no information on which to base the choice of a turning direction, and so this choice is deferred to the sequencer.

Wall-finding was done by turning towards the sonar with the shortest range reading until the shortest reading was given by the forward sonar, and simultaneously moving forward until forced to stop by an obstacle in a hard-obstacle region. This would reliably leave the robot facing the nearest object. When initiated near a wall, the robot would turn towards the wall.

Wall alignment was done by slowly turning the robot until a discontinuity caused by the onset of specular reflection was seen in the range reading returned by the forward sonar. When this procedure was begun while facing a smooth wall, the angle at which the discontinuity occurred was reproducible to better than 1 degree.

Wall following was done by servoing the robot's heading to the reading on a side-facing sonar while moving forward. Although conceptually simple, the actual implementation is complicated by a number of factors.

The main problem is that a straightforward negative-feedback servo loop off a side-facing sonar is unstable if the robot ever turns far enough towards the wall to cause a specular reflection on the side sonar. When this happens, it appears to the robot that the wall is suddenly very far away, and it will continue to turn towards the wall and eventually collide unless the safety module stops it. A similar effect happens when the robot passes an open door or an intersecting corridor.

There are two possible solutions to this problem. The first is to servo to the shortest reading on the side-facing sonar and its two adjacent sonars. The second is to use a model-based estimation algorithm such as a Kahlman filter to compute the distance to the wall. The solution used on Alfred was a model-based estimator (though not a Kahlman filter). The estimator simply rejected any sonar reading that was much greater than the last known distance to the wall. The estimator also kept track of the robot's heading and odometer reading (i.e. the drive motor encoder reading) every time a valid sonar reading was taken. When an invalid sonar reading occurred, the robot turned towards the heading it was on during the last valid reading. If the robot traveled more than two meters without a valid reading the robot stopped.

All of these primitive behaviors were implemented in less than 200 lines of ALFA code.

4.2 Sequencing Layer

Alfred's sequencing code was written in Macintosh Common Lisp version 2.0 (with one exception; see below), using a set of macros that later evolved into ESL [Gat97]. MCL 2.0 is a single-threaded Lisp, which made it impossible to implement multithreaded task management directly. MCL version 3 is multithreaded, and all of the code and infrastructure described in this section have been much improved since Alfred's code was written.

The first contest event required the robot to search an office-like environment for a door, then traverse an obstacle-strewn area to a finish line. The door was opened between one and three minutes after the start of the event, and could be in one of three different locations. The robot was told its initial orientation and the size of the room, but not its initial position nor the locations of obstacles.

Alfred determined its location by wandering randomly for one minute and keeping track of its maximum and minimum positions along the X and Y axes. Wandering was done by augmenting the obstacle-avoidance code with an algorithm for choosing a turn direction when the choice was not clear from the current situation. This must be done with some care, or the robot can get stuck in an infinite loop. Alfred used the following algorithm: when an ambiguous obstacle avoidance situation was encountered the robot would do an angular scan, turning first one way, then the other. The angle of the turn was gradually increased until the robot was able to move forward some threshold distance without triggering the scan. The scan angle was then reset to its initial value. Alfred's wander behavior was actually written in ALFA, although its use of stored internal state to produce discontinuous behavior indicates that it should be considered part of the sequencing layer.

Alfred then attempted to escape from the office by trying each of the three door locations in turn. It would move to the center of the office, point itself towards one of the doors, and turn on the follow-current-heading-with-obstacle-avoidance primitive. It would then wait until it either escaped the office (as indicated by its dead-reckoning position) in which case it headed towards the finish line, or a time limit was reached in which case it tried the next door. This task required no planning.

The second contest was much more interesting and challenging. The robot was put in a maze for which it had been given a complete and accurate map. However, the robot was given no information about its initial position or orientation. The robot's task was to search for a coffeepot hidden in the maze and deliver it to a given destination. The robot was given partial information about the location of the coffeepot. Of course, Alfred had no sensors capable of detecting a coffeepot, so it had to be told when the coffeepot was nearby, but otherwise Alfred completed the task with no cheating.

The key to Alfred's success was a combination of behaviors that allowed for reliable navigation of environments that were rich in walls, like mazes, and some creative representations. In addition to representing the a priori map of the maze in terms of open space, the robot was also given a description of the maze in terms of the wall assemblies that comprised it. (With a little more time the robot could have been programmed to convert from one representation to the other automatically.)

The robot self-localized by first locating a wall. It did this by invoking the wall-finding primitive, and then verifying that it had indeed found a wall rather than an obstacle by attempting to follow it for some distance (2 meters). It then began to follow the wall, turning whenever the wall turned, and keeping track of the sequences of turns. Whenever it made a turn, it

checked to see if the sequence of turns it had made created an unambiguous match with its a priori knowledge of the shapes of the wall assemblies in the maze. (This was done by the deliberator.) As soon as it had a match, the robot knew where it was. It then began a systematic search of the possible locations of the coffeepot, followed by a traversal to the delivery location.

Note that the algorithms in the sequencer make extensive use of internal state (keeping track of which door location is being tried, maintaining records of the robot's position, etc.) but no search or temporal projection.

4.3 Deliberative Layer

The deliberative layer did the matching of Alfred's self-localization sequence to the a priori map, and also planned paths for traveling between locations. Both algorithms were simple exhaustive searches made tractable by the fact that the search space was bounded by the size of the maze.

By the standards of AI, the deliberative layer was trivial and uninteresting, which is precisely what makes the three-layer architecture non-trivial and very interesting. The use of a sequencing layer makes it possible (in fact, easy) to use trivial and uninteresting algorithms to control real robots performing complex tasks.

5. Conclusions

The three-layer architecture arises from the empirical observation that effective algorithms for controlling mobile robots tend to fall into three distinct categories: 1) reactive control algorithms which map sensors directly onto actuators with little or no internal state, 2) algorithms for governing routine sequences of activity which rely extensively on internal state but perform no search, and 3) time-consuming (relative to the rate of change of the environment) search-based algorithms such as planners. The three-layer architecture is based on the premise that algorithms of the first (second) type can provide effective computational abstractions for constructing interfaces to algorithms of the second (third) type. This conclusion has apparently been reached independently by at least three different groups of researchers.

Algorithms of the first and third type can be programmed in conventional programming languages. Algorithms of the second type appear to benefit significantly from specialized languages with sophisticated control constructs. Attempts to construct languages to enforce the constraints imposed on algorithms of the first type have been largely unsuccessful.

In retrospect, in the story of the three-layer architecture there may be more to be learned about research methodology than about robot control architectures. For many years the field was bogged down in the assumption that planning was sufficient for generating intelligent behavior in situated agents. That it is not sufficient clearly does not justify the conclusion that planning is therefore unnecessary. A lot of effort has been spent defending both of these extreme positions. Some of this passion may be the result of a hidden conviction on the part of AI researchers that at the root of intelligence lies a single, simple, elegant mechanism. But if, as seems likely, there is no One True Architecture, and intelligence relies on a hodgepodge of techniques, then the three-layer architecture offers itself as a way to help organize the mess.

The three-layer architecture is by no means the last word in either architectures or organizational tools. It largely ignores, for example, issues like sensor processing, learning, and world modeling. Such algorithms may turn out to fit nicely within the existing structure, or it may prove necessary to extend the architecture to incorporate them. This promises to be fertile ground for future research.

Acknowledgments

Pete Bonasso and Robin Murphy provided extensive and thoughtful comments on an early draft of this chapter. David Miller and Marc Slack provided useful comments on an early draft of section 1. This work was conducted at the Jet Propulsion Lab, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[Agre87] Phillip E. Agre and David Chapman. "Pengi: An Implementation of a Theory of Activity." *Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI), 1987.

[Agre90] Phillip Agre and David Chapman. "What are Plans For?" *Robotics and Autonomous Systems*, vol. 6, pp. 17-34, 1990.

[Angle89] Colin Angle, "Tooth Docs: the Paper", unpublished manuscript.

[Arkin90] Ronald C. Arkin. "Integrating Behavioral, Perceptual and World Knowledge in Reactive Navigation." *Robotics and Autonomous Systems* 6 (1990) 105-122.

[Balch95] Tucker Balch, et al. "Io, Ganymede and Callisto: A multiagent robot trash collecting team." *AI Magazine*, Summer 1995.

[Biber80] C. Biber, et al. "The Polaroid Ultrasonic Ranging System." *Proceedings of the 67th Conference of the Audio Engineering Society*, 1980.

[Bonasso91] R. Peter Bonasso. "Integrating Reaction Plans and Layered Competences Through Synchronous Control." *Proceedings of the Internation Joint Conference on Artificial Intelligence* (IJCAI), 1991.

[Bonasso92] R. Peter Bonasso. "Using Parallel Program Specifications For Reactive Control of Underwater Vehicles." *Journal of Applied Intelligence*, June 1992.

[Bonasso et al. 97] R. Peter Bonasso, et al. Experiences with an Architecture for Intelligent Reactive Agents. *Journal of Experimental and Theoretical AI*, 9(2), 1997.

[Brooks86] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal on Robotics and Automation*, vol RA-2, no. 1, March 1986.

[Brooks89] Rodney A. Brooks. "The Behavior Language User's Guide." MIT AI Lab internal publication.

[Brooks90] Rodney A. Brooks. "Elephants Don't Play Chess." *Robotics and Autonomous Systems* 6 (1990) 3-15.

[Brooks91] Rodney Brooks. "Intelligence without representation." *Artificial Intelligence* 47 (1991) 139-160.

[Connell89] Jonathan Connell. *A Colony Architecture for an Artificial Creature.* Technical Report 1151, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1989.

[Connell91] Jonathan Connell, "SSS: A Hybrid Architecture Applied to Robot Navigation," *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*, 1992.

[Elsaessar&Slack94] Chris E;lsaessar and Marc Slack. "Integrating deliberative planning in a robot architecture." Proceedings of the AIAA Conference on Intelligent Robots in Field, Factory, Service and Space (CIRFFSS), 1994.

[Firby87] R. James Firby. An Investigation Into Reactive Planning in Complex Domains. *Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI), 1987.

[Firby89] R. James Firby. *Adaptive Execution in Dynamic Domains.* Technical report YALEU/CSD/RR#672, Yale University, 1989.

[Firby96] R. James Firby, et al. "Programming CHIP for the IJCAI-95 Robot Competition." *AI Magazine*, Spring 1996.

[Firby&Slack95] R. James Firby and Marc Slack. "Task execution: Interfacing to reactive skill networks." Working notes of the 1995 AAAI Spring Symposium on Lessons Learned from Implemented Architectures for Physical Agents, 1995.

[Gat91] Erann Gat. "Reliable Goal-directed Reactive Control for Real-world Autonomous Mobile Robots." Ph.D. Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1991.

[Gat91b] Erann Gat. "ALFA: A Language for Programming Reactive Robotic Control Systems." *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*, 1991.

[Gat92] Erann Gat, Integrating Planning and Reaction in a Heterogeneous Asynchronous Architecture for Controlling Mobile Robots, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI)*, 1992.

[Gat93] Erann Gat. "On the Role of Stored Internal State in the Control of Autonomous Mobile Robots." *AI Magazine*, Spring 1993.

[Gat94] Erann Gat, et al. "Behavior Control for Robotic Exploration of Planetary Surfaces." *IEEE Transactions on Robotics and Automation* 10 (4) 1994.

[Gat&Dorais94] Erann Gat and Greg Dorais. "Robot Navigation by Conditional Sequencing." *Proceedings of the International Conference on Robotics and Automation* (ICRA), 1994.

[Gat97] Erann Gat. "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents." *Proceedings of the IEEE Aerospace Conference*, 1997.

[Georgeff87] Michael Georgeff and Amy Lanskey, "Reactive Reasoning and Planning", *Proceedings of AAAI-87*.

[Hartley91] Ralph Hartley and Frank Pipitone. "Experiments with the Subsumption Architecture." *Proceedings of the International Conference on Robotics and Automation* (ICRA), 1991.

[Kaelbling87] Leslie Pack Kaelbling. "REX: A Symbolic Language for the Design and Parallel Implementation of Embedded Systems." *Proceedings of the AIAA conference on Computers in Aerospace*, 1987.

[Kaelbling88] Leslie Pack Kaelbling. "Goals as Parallel Program Specifications." *Proceedings of AAAI-88*.

[Kirsh91] David Kirsh. "Today the earwig, tomorrow man?" *Artificial Intelligence* 47 (1991) 161-184.

[Latombe91] Jean-Claude Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[Miller91] David P. Miller, et al. "Reactive Navigation through Rough Terrain: Experimental Results." *Proceedings of AAAI92*.

[Nilsson80] Nils J. Nilsson. *Principles of Artificial Intelligence*. Palo Alto: Tioga. 1980.

[Noreils90] Fabrice Noreils. "Integrating Error Recovery in a Mobile Robot Control System." *IEEE International Conference on Robotics and Automation*, 1990.

[Nourbakhsh et al. 93] Illah Nourbakhsh, et al. "The Winning Robots from the 1993 Robot Competition." *AI Magazine*, Winter 1993.

[Payton86] David Payton. "An Architecture for Reflexive Autonomous Vehicle Control." *Proceedings of the International Conference on Robotics and Automation* (ICRA), 1986.

[Payton90] David Payton, J. Kenneth Rosenblatt and David Keirsey, "Plan-Guided Reaction," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 20, pp. 1370-1382, 1990.

[Pell et al. 96] Barney Pell, et al. "A Remote Agent Prototype for an Autonomous Spacecraft." *Proceedings of the SPIE Conference on Optical Science, Engineering, and Instrumentation*, 1996.

[Rosenblatt89] J. Kenneth Rosenblatt and David W. Payton "A Fine-grained Alternative to the Subsumption Architecture." Proceedings of the AAAI Stanford Spring Symposium Series, 1989.

[Schoppers87] Marcel Schoppers,=. "Universal Plans for Reactive Robots in Unpredictable Domains." *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1987.

[Simmons90] Reid Simmons. "An Architecture for Coordinating Planning, Sensing and Action." *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[Simmons95] Reid Simmons. "The 1994 AAI Robot Competition and Exhibition." *AI Magazine*, Summer 1995.

[Soldo90] Monnett Soldo. "Reactive and Preplanned Control in a Mobile Robot." *Proceedings of the International Conference on Robotics and Automation* (ICRA), 1990.

[Suchman87] Lucy Suchman. *Plans and Situated Action*. Cambridge University Press, 1987.

[Wilcox87] W. H. Wilcox, et al., "A vision system for a mars rover," *Procedings of SPIE Mobile Robots II*, vol. 852, 1987.