# Data Persistence: A Design Principle for Hybrid Robot Control Architectures

**Mithun Sheshagiri**
Maple Research Group,
ECS-002, UMBC,
Baltimore, MD-21227, U.S.A.
Email: *mits1@cs.umbc.edu*

**Marie desJardins**
Maple Research Group,
ECS-002, UMBC,
Baltimore, MD-21227, U.S.A
Email: *mariedj@.cs.umbc.edu*

**Abstract**

It has been widely accepted that intelligent robots operating in dynamic environments should be built using a combination of two very different approaches: reactive and deliberative architectures. The use of only one of these approaches leads to sub-optimal performance when non-trivial tasks are to be performed by robots in dynamic environments. Typical hybrid architectures consist of several layers with one reactive layer and several other layers with varying levels of deliberativeness. This paper discusses existing hybrid architectures in terms of the flow of control across various layers and their limitations. The paper also proposes a new architecture, Layering Using Data Persistence (LUDaP) that uses the notion of *data persistence* to guide the design of the layers in the architecture.

## 1. Introduction

The field of AI started out as an attempt to replicate human intelligence. A major chunk of the AI community worked on systems that could solve problems and learn. This interest was reasonable and justified since our ability to solve problems and learn is at the core of our intelligence. These early systems worked by operating on elaborate representations of the world and can be characterized as *deliberative* systems. The negative aspect of this trend was lack of attention given to other faculties of human intelligence, which dealt with seemingly un-intelligent things like motion.

For robot builders, using deliberative systems in robot control systems became an obvious choice due to the popularity and availability of these deliberative systems. However, robots that made use of deliberative framework were excruciatingly slow. This can be directly attributed to the nature of deliberative systems, which are intensive in terms of both memory and computation.

In the nineties, [Brooks, 1996] proposed some radical new ideas that changed the way in which autonomous robots were built. Brooks suggested that for a dynamically changing world, a close coupling between percepts and actions was the key to building successful robots. According to him, having a representation of the world and committing to it is a flawed approach [Brooks, 1991]. Architectures that do not make use of any representation of the world, relying instead on rapid percept-action cycles to achieve their objectives, are termed *reactive*. This paradigm can be used to build very successful but simplistic robots. Due to the absence of representation, purely reactive robots find complicated tasks (tasks that need to be performed in a specific order or that require the system to maintain an earlier state) difficult to achieve.

Intuitively we can infer from the complementary advantages and disadvantages of reactive and deliberative approaches that successful robots that can achieve complex tasks should be built using a combination of both of these approaches. The paper proposes a hybrid architecture that uses *persistence of data* in the system to characterize the layers that form the hybrid architecture.

### 1.1. Deliberative Systems

Deliberative systems have the following properties:

1. They have an explicit representation of the world and operate on this representation to achieve tasks.

- Advantage:
    - Complex goals can be specified in terms of this representation.
    - Learning can be incorporated.
- Drawback: There is no optimal way to represent the world. This is because of the "what-if" or qualification problem; it is impossible to represent even a simple world completely by enumerating.

2. The system gets percepts from the world. It modifies its representation of the world using these percepts and computes an appropriate action. This is called the percept-action cycle. One percept-action cycle could last several seconds or minutes.

- Drawback: If the world in which the system operates changes rapidly, then long percept-action cycles are of very little use. This is primarily because the action arrived at after the percept-action cycle might be inappropriate due to changes in world after the percept-action cycle starts.

### 1.2. Reactive Systems

Reactive systems have the following properties:

1. Very limited or no representation of the world.

- Advantage: Reduces the amount of resources required for computation.
- Drawback:
    - Difficult to specify complex goals.
    - Difficult to incorporate learning modules.

2. The percept-action cycle is very short.

- Advantage: Domains in which the environment is constantly changing can be handled.

There are several implementations of hybrid architectures, Section 2 looks at a couple of them. In Section 3, we discuss about our framework for building hybrid systems. Section 4 and Section 5 talk about our implementation and experimental results.

## 2. Related Work

Hybrid architectures represent an attempt to incorporate the best of both worlds. They exploit the deliberative method's inherent potential to achieve complicated tasks and to incorporate learning, while providing the fast and responsive behavior of reactive methods. We next briefly describe two systems that have integrated a deliberative planner and a reactive layer.

### 2.1. ATLANTIS

ATLANTIS [Gat,1992] was designed to control autonomous robots capable of performing multiple tasks in a real-time, noisy and unpredictable environment. The various layers in the system operate asynchronously and the computational structure of each of these layers is very different (heterogeneous). The system is made up of three layers: a controller, a sequencer and a planner. The controller is responsible for all primitive activities and forms the reactive layer of the system. The sequencer sequences primitive operations and deliberative computations, providing an interface between the lower reactive and upper deliberative layer. An important feature of ATLANTIS is that the upper layer merely advises on what needs to be done. The decision whether to take up the advice is left to the sequencer.

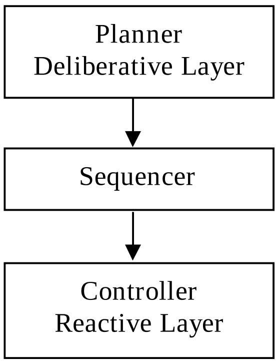

Fig. 1: Architecture of ATLANTIS

### 2.2. AuRA

The architecture of the Autonomous Robot Architecture (AuRA) [Arkin *et al.*, 1997] is more elaborate than that of ATLANTIS, consisting of three layers: two deliberative layers and a reactive layer. The bottom reactive layer is based on Schema theory as developed by Arbib [Arbib, 1992] and implemented by Arkin [Arkin, 1995]. The main difference between ATLANTIS and AuRA• apart from the number of layers• is the way in which the deliberative layers are put into action. In AuRA, the upper layers are not activated unless a failure is detected. Lack of progress constitutes failure. The upper layers are activated one layer at a time until the problem causing the failure is resolved. Unlike ATLANTIS, AuRA incorporates several forms of learning techniques. Although learning primarily occurs in the deliberative layer, it can be incorporated into the reactive layer.

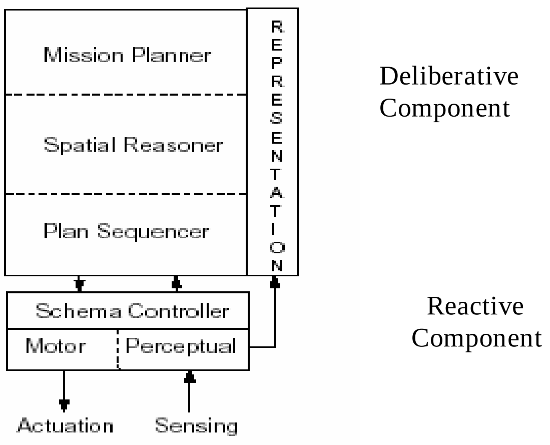

Fig. 2: High Level AuRA Schematic (This figure has been directly reproduced from [Arkin, 1997])

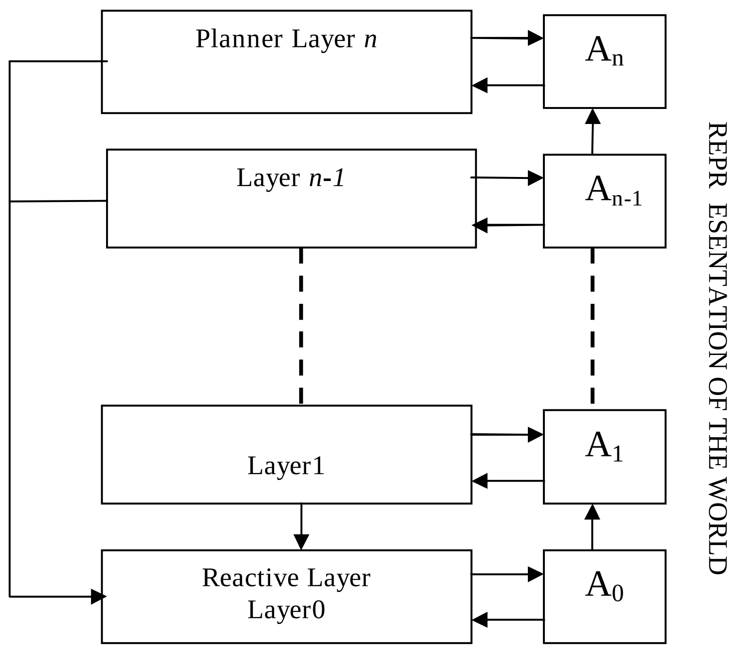

Fig.3 Proposed LUDaP Architecture

## 3. Layering Using Data Persistence (LUDaP)

Although both ATLANTIS and AURA do a good job of defining interfaces between the deliberative layer and the reactive layer, the methodology adopted for their design is very domain-specific. These are very successful working systems but it is difficult to port their architecture to a new domain. Moreover it appears that the partitioning of layers is done based on the amount of time required for various tasks. Computationally intensive tasks move upwards. Although this is a necessary condition, we believe that this is not sufficient to build efficient architectures. We propose a *persistence-based* approach to layer design in hybrid systems, where the functionality and reasoning of each layer is characterized by the degree of persistence of the data that is processed by that layer. There are several issues that need to be resolved when building persistence based hybrid architectures.

### 3.1. Data Persistence

Data which is not likely to change over some duration of time is termed persistent data. The *degree of persistence* of a data item is directly proportional to the duration during which the data does not change. Chunks of data that change continuously but gradually can also be thought of as persistent data. In this case, the degree of persistence is proportional to the rate of change.

We define the following classes of persistent data:
1. Data which is persistent by nature.
2. Data formed by abstraction.
3. Predictable data.

We now define these three classes of data, and illustrate them using a scenario in which robot is trying to navigate through a room and has to visit a specific set of points as part of its goals. We assume in this scenario that's the robot is self-powered by a battery pack.

### 3.1.1. Persistence by Nature

In almost all domains, it is possible to determine parts of the representation of the world (data) that are static or that change slowly over time. The rate at which each of these chunks of data change may vary. Hence we say that the *degree of persistence* varies for these chunks of data. For example, in the room navigation domain, the position of the robot ***position(x,y)*** is constantly changing and therefore has a very low degree of persistence *by nature*. The fact that the robot has visited a particular point, ***visited(x,y)***, does not change once established; therefore, this fact has a very high degree of persistence by nature. At the same time, the relations that characterize the points that have not been visited by the robot, ***unvisited(x,y),*** have a lower degree of persistence as their status is likely to change in the future.
As is clear from the above example that isolating this class of persistent data requires intimate knowledge of the domain. In this example, we may not know the precise degree of persistence of the relations ***visited(x,y)*** and ***unvisited(x,y)***, but we can make the following qualitative statements based solely on the semantics of the domain.

***d.o.p[visited(x,y)] > d.o.p[unvisited(x,y)] > d.o.p[position(x,y)]***
where ***d.o.p= degree of persistence***

### 3.1.2. Data Formed by Abstraction

Designing any robot architecture involves data abstraction. Here we define data abstraction as the formation of complex forms of data from simpler forms of data. When abstract data are formed, they acquire certain properties: in particular, reasoning using abstract data may take less time than using the raw data; on the other hand, forming and updating these abstract data structures incurs computational overhead. These complex forms of data sometimes acquire a higher degree of persistence in cases where they aggregate states or properties of the world. For example, in the robot navigation domain, we could form a spatial abstraction that groups together individual locations and dividing the room into a grid and characterize the position of the robot in terms of the blocks in the grid. In this case,
***d.o.p[block(u,v)] > d.o.p[position(x,y)]***
where ***(u,v)*** is the coordinate of the block in the grid.
The map of the room is an abstraction formed from the position (low degree of persistence) of the robot.

### 3.1.3. Predictable Data

Most domains also have data that are known to change in a predictable way. These forms of data are persistent by virtue of the fact that we can reason about them using their predictable nature, with some associated uncertainty. The weight of this uncertainty factor increases with the time since the data item was last directly observed. Therefore, the degree of persistence of this form of data decreases over time until its true value is reinforced by the percepts received by the robot. To illustrate this in the robot navigation domain, we define power-on time as the time remaining till battery failure. The power-on time of the robot can be estimated using the power drawn by the robot per unit time. We categorize the power-on time as persistent, based on its predictable nature. The power-on time of the robot can be accurately estimated at any given time and used for deliberation without significant loss of accuracy. In the case where operations performed by the robot consume different amounts of power, an uncertainty value can be associated with the estimate. Note that associating an uncertainty value decreases the degree of persistence.

## 3.2. Need for Layering

The robot needs to be responsive (reactive) to perform tasks in a dynamic environment. Including complex tasks into the reactive module will slow it down. Therefore complex (computationally intensive) tasks should be assigned to separate layers, i.e., moved to the upper layers. We believe that, to make the design more efficient, the data used for performing these tasks should be chosen using the concept of data persistence.

A deliberative system maintains a representation of the world. On receiving percepts, it reasons using this representation of the world and produces actions that it believes are beneficial in achieving its goal. Consider a system that believes the state of the world to be ***A***. On receiving percepts, it uses ***A*** for reasoning and concludes that execution of an operator ***X*** moves it closer to its goal. It assumes that ***A*** does not change during the reasoning process. Once a suitable operator is determined, one of the following approaches can be used: The operator ***X*** can be executed directly, or the operator can be executed contingent on testing whether the data that was used for selection of operator has changed. In either case, deliberation is likely to fail when the environment is highly dynamic. If the selected operator is executed without checking the state of the world, then there is a high probability of executing misappropriate actions. If a test is performed before executing the action and the condition fails to be met, then the result of the deliberation is rendered useless.

Fig.4 (a) shows a scenario where deliberation is likely to fail. If it can be assured that the data is persistent throughout the computation, then all the above issues disappear. What we need is the scenario in fig. 4 (b) where deliberation is useful. We claim that by classifying data using data persistence as the guiding factor and constructing layered reasoning systems based on this criterion, we can build an architecture in which the selection and the execution of appropriate actions at each layer is possible.

In a dynamic world, some part of ***A*** (the representation of the world) keeps changing; as a result, the theory of data persistence must be applied by dividing ***A*** into smaller chunks of data. These chunks of data consist of data items that have degrees of data persistence within a small range. Data structures from different chunks should have a substantial difference in their degree of persistence. This division of data is an important factor during the design of the system. The approach adopted by us for our implementation was to enumerate the possible data items and use a combination of our intuition and domain knowledge to group them into chunks. We are exploring theoretical methods that would allow a more rigorous characterization of the data persistence within a given domain.

$A \bullet A_0 \; U A_1 \; U \; A_2 \; U \; A_3 \; U \; \ldots\ldots A_{n-1} \; U \; A_n$

*Data items in* $A_{i+1}$ *are more persistent than data items in* $A_i$

*Data items in* $A_{i+1}$ *can be derived from data items in* $A_0 \ldots\ldots\ldots A_{i-1}, A_i$

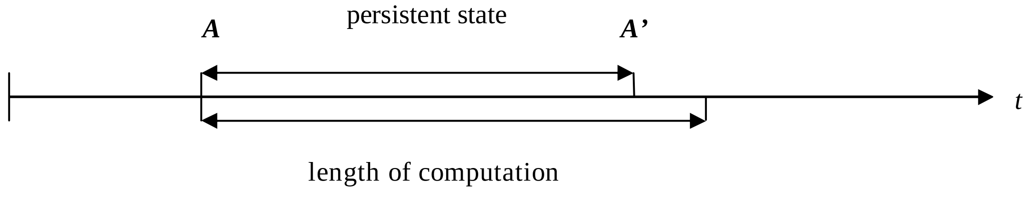

Figure 4 (a)

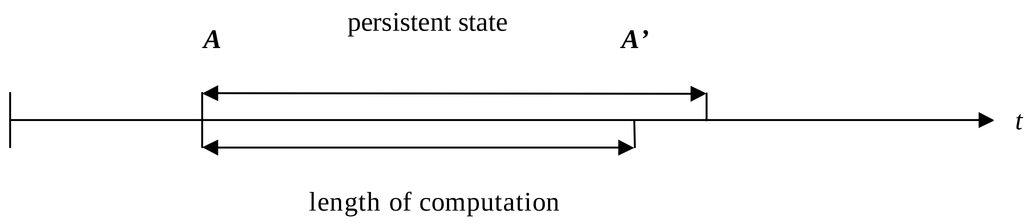

Figure 4 (b)

### 3.3. Number of Layers

The number of layers in AuRA and ATLANTIS is fixed. The number of layers was most likely determined after a thorough analysis of the specific problem being addressed. It is difficult to fit different tasks into this fixed number of layers.

In LUDaP, a layer must be devoted to each $\boldsymbol{A_i}$. Each layer should strictly handle data assigned to it, i.e., ***Layer**$_i$ operates using* $\boldsymbol{A_i}$ only. An important thing to note is that each layer depends on one or more of the bottom layers with respect to only *data*. A layer's working is not dependent on the *functioning* of the lower layers, i.e., the upper layers do not rely on which primitives are used or which actions are selected by the lower layers. This is a significant advantage and constraint: the final system can be built as a parallel processing system consisting of multiple processors each handling a separate layer. A single processor could implement such a system by using multiple threads with each thread representing a layer. Our implementation uses the latter approach. Independence of individual layers also makes LUDaP scalable: additional layers can be added as long as the data-sharing mechanism doesn't become a bottleneck.

### 3.4. Activations of Individual Layers

In ATLANTIS, all layers operate synchronously: the deliberative layer merely advises the sequencer to follow specific commands. In AuRA, when a failure is detected, control is moved to an upper layer; the lower layer waits until the problem is resolved. From the moment the failure is detected until the failure is resolved, the robot does little to achieve its goal. In this phase the robot is prone to the dangers of a dynamic environment.

At any given time, the layers built using LUDaP could be in either of the two stages: checking for preconditions for the execution of actions or directing the lowermost reactive layer to execute a specified action. We combine these two terms and refer them as computing (to simplify explanation). In LUDaP, depending on the implementation, the upper layers may or may not be computing all the time. In a multi-processor environment all layers could be computing at the same time; and in a single processor environment, layers are logically computing concurrently, but physically they compute only when they are allotted a CPU time slot.

In LUDaP, the reasoning in the upper (deliberative) layers uses certain preconditions to control the lowermost layer. Each layer has a different set of preconditions; the frequency with which the conditions in each of the layers are satisfied reduces as one moves up the hierarchy. This can be explained as follows: after a layer directs the reactive layer to execute an action it starts checking for the preconditions again. For the next set of preconditions to be satisfied, the value of one or more data items in that layer has to change. Since change is intimately related to degree of persistence, the frequency with which the preconditions are satisfied is directly proportional to the degree of persistence. The reactive layer, with its short percept-action cycle, forms the most critical component in a dynamic environment. Therefore an effort should be made to supply this layer as much processing power as possible. The LUDaP architecture gives the designer the flexibility to satisfy this constraint. By forcing upper layers to operate on data with higher degree of persistence, we can justify the allocation of fewer CPU cycles to the upper layers.

## 4. Current Implementations

Currently LUDaP is being built on the TeamBots© [Balch, 2000] robot simulator. This Java-based simulator provides a library of primitives built on schema theory.

The *herding* scenario involves two types of robots: *Pursuers* and *Evaders*. Pursuers have radars to detect walls and have visual sensors that detect evaders within a specified range. Evaders have radars to detect the presence of obstacles and pursuers. As soon as a pursuer is detected by the radar, evaders move away from them. Herding involves searching for an evader robot and directing it towards a specified area. The evader, meanwhile, tries to move away from the pursuer. Tracking the evader, avoiding obstacles, and moving to a specific point involve dynamic data and therefore are handled by the reactive layer (*layer0*).

The pursuers switch between various primitive behaviors (like searching or directing an evader) using a finite-state-automaton model. A robot is always in one of the several states. It changes state based on specific percepts. For example, a robot continues to stay in the search state until it encounters an evader. As soon as it finds an evader in its range, it starts patrolling the evader to the patrol region. The state switching can also be done by upper layers (this is how they control the reactive layer).

The pursuer is not capable of coordinating its activities with other pursuer robots, using the reactive layer alone. For example, multiple pursuer agents could be following the same evader. We built an additional layer *(layer1)* that checks for the presence of other pursuers in nearby vicinity. This layer also checks the state of the nearby pursuers. The information required to perform these checks is gathered from messages that are broadcasted by all pursuers. The message contains information about the state, position of the robot, and the number of evaders in the range of the pursuer. Because communication is an expensive (time-consuming) operation, the content of the messages should possess a high degree of persistence, where the required degree of persistence is directly proportional to the time required for transmitting the message. State information is more persistent than the position of the robot and therefore assigned to a higher layer. The position information is used to reason about the presence of the robot in an *area*, therefore is a form of persistent data.

The topmost layer *(layer2)* forms a map (grid) of all visited areas and directs an idle robot to locations that have not been visited by any robot for some time. The map data structure is persistent by virtue of the fact that rate of change of the map is very slow and therefore qualifies it as a data item with high degree of persistence. Here the map data item is incorporated into *layer2* rather than *layer1*. Looking for vacant spaces in a map is also computationally more expensive than looking for other pursuers. By incorporating the map functionality into *layer2,* we keep *layer1* responsive.

Once all evaders have been detected, the number of evaders in the system will not change; therefore, this type of data is highly persistent and can be used by the upper layers.

## 5. Experimental Results

*Layer1* helps pursuer from crowding around a single evader and directs them to locations where they is likely to find more evaders. *Layer2* directs idle pursuers to places that have not been covered by any pursuer for some time. Both these layers enable pursuers to discover evaders. Our experiments capture the average number of evaders discovered with the following variations:
1. *layer0* (only reactive)
2. *layer0 + layer1* (with some reasoning)
3. *layer0 + layer1+layer2* (all layers functioning)
The experiments were run five times.

| | The average number of evaders detected | | |
|---|---|---|---|
| Cycle no. | layer0 | layer0+layer1 | layer0+layer1+layer2 |
| **1** | **1.6** | **26** | **25** |
| **2** | **1.6** | **22.6** | **28.20** |
| **3** | **1** | **24** | **31** |
| **4** | **1.4** | **14.6** | **22.60** |
| **5** | **1.4** | **24.8** | **22.80** |
| Avg. | **1.4** | **22.4** | **25.92** |

Our results reinforce the belief that adding deliberativeness significantly increases performance. Although adding another layer improves performance, it is not clear if this can be statistically justified. We therefore conclude that deliberation is necessary to improve performance and a principled methodology for designing and integrating additional layers is highly desirable. LUDaP is a step in this direction.

## 6. Future Work

The current implementation consists of three layers and does not incorporate a truly deliberative (planner like) layer. The existing implementation is therefore a partial hybrid architecture. The mechanism to classify data into chunks is currently driven mostly by intuition and domain knowledge. Development of a more principled and formal approach to address this issue is our most important goal. The mechanism used for controlling the reactive layer by the upper layers• specifically resolution of conflicts is another issue we are working on.

## 7. Conclusions

This paper proposes a new framework for building hybrid robot architectures. We introduce the concept of data persistence which forms the guiding factor for designing hybrid control systems using this framework. We borrow some ideas from existing hybrid architectures and introduce some new ideas.

## References

[Arbib, 1992] Arbib, M. “Schema Theory”, in Encyclopedia of Artificial Intelligence, 2nd ed., (S. Shapiro, ed.,) Wiley, 1992, pp. 1427-1443.

[Arkin and Balch, 1997] Ronald C. Arkin, Tucker Balch, “AuRA: Principles and Practice in Review”, JETAI, volume 9, pp 175-189, 1997

[Arkin, 1995] Arkin, R.C., "Reactive Robotic Systems", Handbook of Brain Theory and Neural Networks, ed. M. Arbib, MIT Press, pp. 793-796, 1995.

[Brooks, 1996] Rodney A. Brooks, “A Robust Layered Control System for a Mobile Robot”, IEEE Journal on Robotics and Automation, vol. RA-2, no. 1, March 1996.
[Brooks, 1991] R.A. Brooks. “Intelligence without Representation.” Artificial Intelligence, Vol.47, 1991, pp.139-159

[Gat, 1992] Erann Gat, “Integrating Planning and Reacting in a Hetrogenous Asynchronous Architecture for Controlling Real-World Mobile Robots”, Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI), 1992.

[Balch, 2000] TeamBots© (www.teambots.org): Simulation and real robot execution environment. Written in Java. Developed at CMU and Georgia Tech.