Architecture Overview

Document Revision 1.02

Samuel M. Smith PhD

242 East 600 North
Lindon
Utah 84042-1662
T: 801.766.3527
F: 801.766.3529
E:
W:

2013/10/25

Table of Contents

1. Introduction

The principle motivation for ioflo is to make it much easier to implement automated reasoning software that achieves higher levels of *meaningful* intelligent autonomy in automation, autonomous agent, robot or other related software systems. The key word here being *meaningful*, that is, intelligent autonomy thats purposeful and useful, instead of just being clever or novel. It is our opinion, based on years of observation and participation in autonomous vehicle and automation system software development, that the primary barrier to higher levels of meaningful intelligent autonomy is *programmer capacity* not *processor capacity*. It is primarily an issue of the *apparent complexity* and the *perceived risk* to the programmer.

We define *apparent complexity* as the number of dependencies between elements of a system that a programmer must manage in order to make meaningful enhancements to the software functionality. In contra-distinction, we define *real complexity* as the total number of dependencies between elements of a system. This definition is coherent with standard metrics for software complexity. The ratio of *apparent complexity* to *real complexity* for a given system architecture is a measure of the "goodness" of the architecture's design. A relatively low ratio indicates a better architecture. Indeed, achieving low apparent complexity relative to high real complexity is the primary goal of ioflo.

Similarly, the *perceived risk* is the peril or exposure to loss the programmer faces when attempting to add meaningful enhancements to the autonomous vehicle software.

The perceived risk includes 1) the estimated programmer time needed to develop and test any code changes, 2) the additional parameters needed to manage mission configuration, as well as 3) the exposure to failure or loss from bugs or other unintended consequences of the changes. Because of the hazardous and unpredictable nature of at-sea operations, the exposure to failure and loss for sea-going autonomous vehicles is very real. Adding higher levels of intelligent autonomy increases the real complexity of the system and unless carefully designed also increases the apparent complexity. High levels of apparent complexity increase the perceived risk because failures are more likely to occur when the programmer cannot successfully manage all the dependencies.

Therefore, with good reason developers of autonomous vehicle software are fearful, cautious, and circumspect when it comes to adding more intelligent autonomy capability. Indeed, developers will go to great lengths to find workarounds that achieve mission objectives without using high levels of intelligent autonomy. Thus, the actual levels of intelligent autonomy found in sea-going unmanned vehicles is quite low despite the many well known methods that could be used given the usable processor capacity. Given the risks involved and the difficulty in understanding the constraints associated with all the vehicle sub-systems (especially with regard to safety), the process of integrating new capability onto a real platform is very complex.

Architectures that scale well with respect to complexity, keep apparently complexity low as real complexity increases. Recall that we defined real complexity as the actual number of dependencies. Whereas apparent complexity as the number of dependencies in the system that the programmer must manage, track, or understand in order add capability. Thus good architectures manage or hide dependencies so the programmer can make changes without much risk. Inevitably, however, adding certain capabilities will break an architecture such that the new dependencies are no longer hidden or well managed and development then hits a complexity *barrier* which it usually cannot cross without changing the architecture. The drawing below illustrates how autonomy architectures map real complexity onto apparent complexity and the resulting apparent complexity barrier.

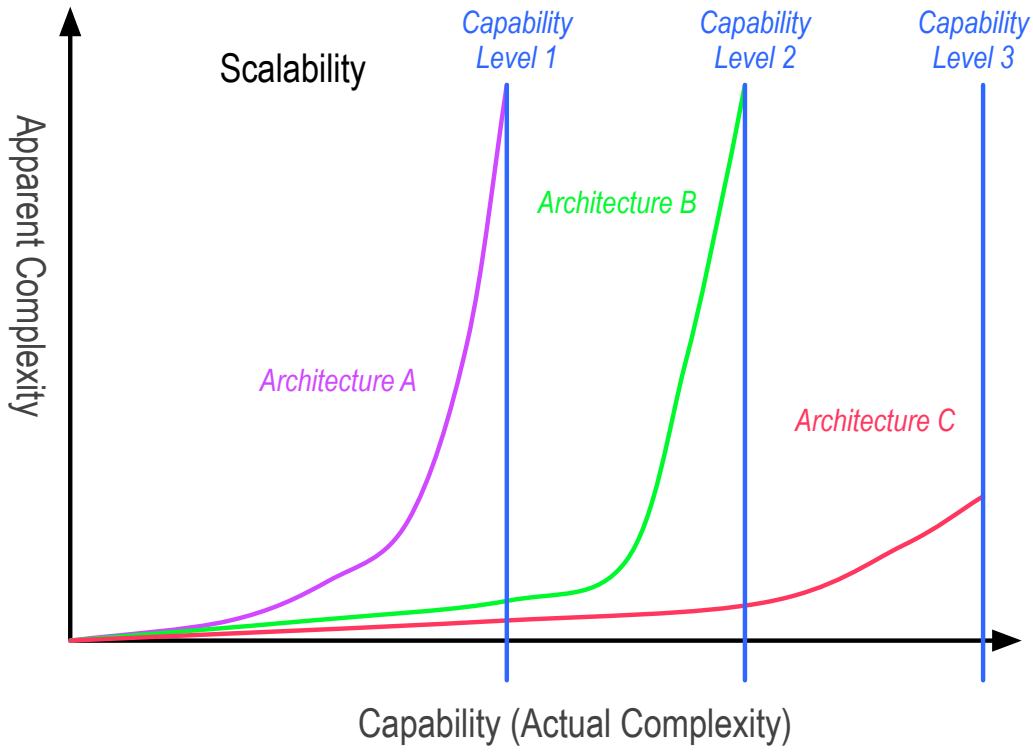


Fig.1.1: Real to Apparent Complexity Mapping. Complexity Barrier.

Much of the "intelligent autonomy" research done by others especially in in non-underwater, academic, or laboratory settings or settings is not meaningful because they do not ever have to address the complicating constraints or "real" world operation. Likewise, the risk of change, in these setting is so much lower that perceived risk is not a significant constraint on programmer capacity. Consequently many of these efforts have not yet overcome the apparent complexity barrier; they haven't had to stress their architectures enough to reach it. Many intelligent autonomy architectures do not scale well enough with regard to complexity to allow them to cross this barrier. Having crossed this barrier several times for different fielded autonomous vehicle platforms, we observed a convergence in the feature set of a practical meaningful intelligent autonomy architecture. Indeed, apparent complexity is a key design issue that must be managed in order to cost effectively add meaningful intelligent autonomy capabilities.

Thus the first step in increasing the level of meaningful intelligent autonomy is to provide a framework that significantly reduces apparent complexity and perceived risk to the programmer for adding meaningful intelligent autonomy. The primary motivation for ioflo is to help provide just such a framework. The ioflo framework is just one piece of the puzzle but the core foundation upon which applications are built.

2. Managing Complexity Through Dependency Reduction

One key to complexity management is dependency reduction, that is, reduce the number of dependencies between components in the system so that changes do not cascade to a point where the time and the effort required to make a change is prohibitive. Since dependencies arise from interaction or exchange of information and data between components of the system, the mechanism

used for information exchange between software components will have the greatest effect on the number and type of dependencies.

Consider, for example, a *one-to-one* information exchange between two software components, *A* and *B*. In order for either *A* or *B* to exchange new information with the other, one or the other of *A* and *B* must know three things; 1) the *identity* of the other component and, 2) the information exchange *interface* with the other component, and 3) the new *information* to be exchanged. A dependency between two items means that anytime one item changes, the other item also changes. Thus, a one way transfer of information from object A to B involves managing three potential dependencies, these are: *component identity* (or *identity* for short), *exchange interface* (or *interface* for short), and *new information* (or *information* for short). Since the purpose of the exchange is to transfer new information from one to the other, the *information* dependency will always be visible. The trick is to hide as much as possible the other two potential dependencies.

The key to managing both the *identity* and *interface* dependencies is controlling how the associated information exchange is initiated. There are two ways to initiate an exchange of new information from *A* to *B*:

1) *A pushes* the new information to *B* or,

2) *B pulls* the new information from *A*.

In either case of *push* or *pull*, if the *interface* between *A* and *B* changes then both *A* and *B* must be changed, so a changeable interface always increases apparent complexity. By change we mean something the software programmer must manage such as a code or configuration modification and therefore contributes to apparent complexity.

In the case of *push*, a change to the identity of *A*, where *A* is replaced with a different software component or is moved to a different location or address, *does not* require a change to *B*, but a change to the identity of *B does* require a change to *A*. In this case, when the identity of B changes, the requisite change to *A* may be to *A*'s configuration or its code or something else to enable *A* to know the new identity of *B*. Depending on how A is built or linked, a change to A's code or configuration may result in a change to A's identity (location) which would then propagate changes to any other components that had identity dependencies on A. To summarize, in the case of *push,* the identity dependency is one way, in the sense that, *A* is dependent on *B*, but *B* is not dependent on *A*. So if we can guarantee that the identity of B will always be fixed then the dependency is effectively hidden, thereby reducing apparent complexity.

Correspondingly, in the case of *pull*, a change to the identity of *A*, *does* require a change to *B*, but a change to the identity of *B does not* require a change to *A*. Thus, in the case of *pull*, the identity dependency is still one way but in the opposite direction, that is , *A* is not dependent on *B*, but *B* is dependent on *A*. So if we can guarantee that the identity of A will always be fixed then the dependency is effectively hidden, thereby reducing apparent complexity.

Likewise if the direction of information exchange is reversed, that is, the new information goes from B to A, then the component initiating the push and pull respectively changes as well as the direction of the dependencies. One way to keep track of this is give a label, such as downstream or upstream, to the direction of a dependency with respect to the direction of the information exchange between any two components. A downstream identity dependency of component *A* on component *B,* means that for new information flowing from A down to B, A is dependent on B, or in other words, a change to

the identity of B requires a change be made to A. Symmetrically, an upstream identity dependency of component B on component A, means that for new information flowing from A down to B, B is dependent on A, or in other words, a change to the identity of A requires a change be made to B.

The relative apparent complexity of O*ne-to-one* exchanges may be considered symmetric with respect to push or pull, that is, if the identity of either A or B or the direction of information flow may change then the apparent complexity of push or pull is the same, both cases have the same number of potential apparent dependencies. Only in the case that, the information flow direction is fixed and the identity of one of A or B is fixed then, depending what is fixed, one of push or pull will have lower apparent complexity.

Now consider how the complexity changes with respect to *one-to-many* or *many-to-one* information exchanges.

First consider *a one-to-many* exchange of new information from component *A* to all of components *B*, *C*, *D, ...* .

In the case of push, if the identity of either *B*, *C*, *D, ...,* changes then A must change. But A may change without requiring any change to B, C, D, ... The more components B, C, D, ... however, the more likely that one of them will have to be changed at some point, thus making it difficult to require that all of B, C, D, ... remain fixed. Thus, the more components B, C, D, ... that A is dependent on makes it more likely that A will have to change thus propagating changes upstream of A to any components dependent on A.

Alternately, in the case of pull, the identities of all of B, C, D, ... could change without A changing. However, if the identity of A changes then all of B, C, D, ... must change, which would cascade changes to any components dependent on all of B, C, D, ... Thus it would be advantageous if A could be fixed. Consequently, pull is likely to have lower apparent complexity versus push since only A has to be fixed with pull.

Next consider a *many-to-one* exchange of new information from any of components W, X, Y, etc. down to component Z. A many to one exchange is more complicated, in that there are many items of new information being exchanged. So the role of Z is important. Consider the case where Z is a selector or consolidator of the new information it receives. In other words, Z sends out only one item of new information to Z's downstream components when it receives multiple items of new information from upstream W, X, Y, ...

In the case of push, any of the identities of W, X, Y, ... may change without requiring a change in Z as long as Z can accommodate a variable number of upstream components. But a change in the identity of Z would require a change to all of W, X, Y, ..., which would propagate to all the components dependent on W, X, Y, ... Thus it would be advantageous if Z could be fixed. Consequently, push is likely to have lower apparent complexity versus pull since only Z has to be fixed with push.

These various dependency relationships are shown in the figure below.

Going downstream, if the *one* could be fixed, one-to-many exchanges should be pulled and many-to-one should be pushed to reduce apparent complexity. But if we consider the case where information exchanges are chained together then we have a problem since the fixed recipient of one exchange would have to be the unfixed sender of the next exchange. The solution is to separate the identity of the component from the identity or location of the information exchange interface and then devise a scheme such that the identity of the information exchange interface is always fixed but the

components using the interface can change identities. One well known successful way to accomplish this with what is called a publish subscribe system.

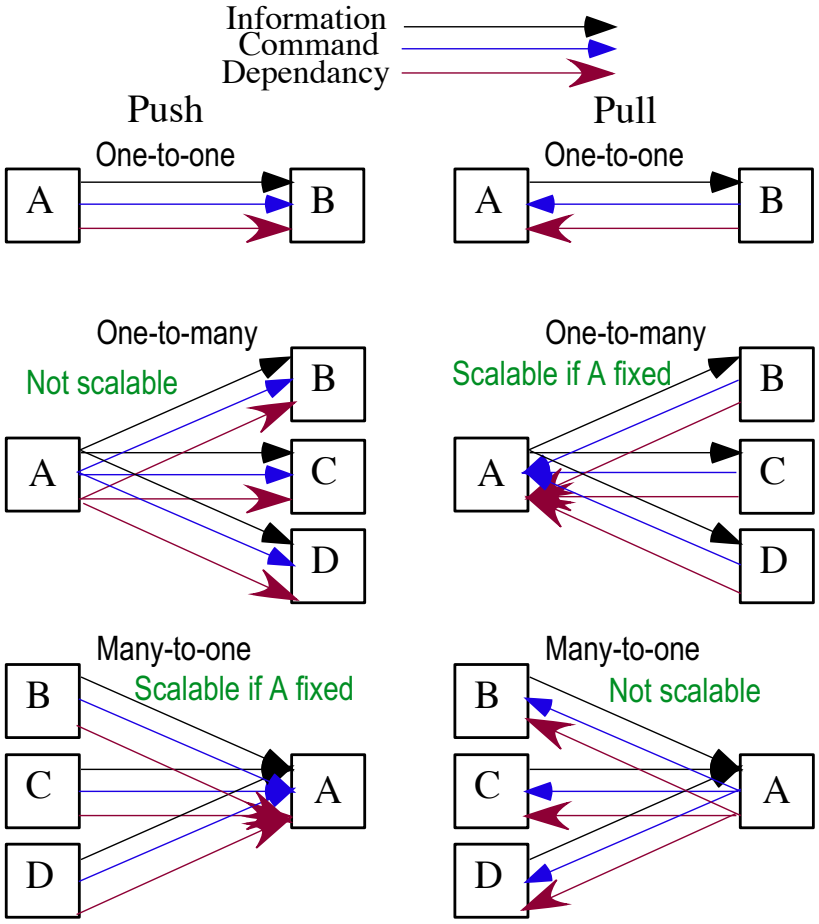


Fig.2.1: Dependency Types

2.1. Pub/Sub Dependency Decoupling

Publish and subscribe (also called producer-consumer) systems, get rid of both the interface and identity dependencies by using what could be called a shared database, an information registry, a data exchange, or shared data store. We use the term Shared Data Store or Store for short. Each item of information in the Store has a public identity that is fixed. For a given item of information, a component may be either a publisher or subscriber. A publisher sends or writes new data to the item. A subscriber retrieves or reads data from the item. Both the identity of and the interfaces to the Store are fixed for a given item of information thereby nulling out the dependencies. All dependencies go from components to the Store. Information flow is always either pushed from components to the Store or pulled by components from the Store, but not between components directly. Consequently, changing or replacing a component does not require changing any other component so apparent complexity is minimized. Some call this feature replacement independence. It is a measure of the extent that connections be changed without the components needing to know about the change. High degrees of replacement independence means that a component can be replaced with a minimum of other changes in the system. Indeed, we see a convergence of best practices in complex software systems, especially distributed ones, to publish-subscribe like architectures (DoD GIG, DDX, RTPS,

DDS, OMG Corba, Etc). We believe the attraction of pub/sub systems arises from this unstated or little understood lowering of apparent complexity via dependency reduction.

Recall that from the perspective of data or information flow there are two types of dependencies, downstream and upstream where down is on the output side of a component's data flow and up is on the input side of a component's data flow. A pub/sub Store removes all downstream dependencies since any publisher does not need to know who the subscribers are to the data in order to publish it. A pub/sub data share, also removes upstream dependencies for one-to-many exchanges, where there are multiple subscribers to the same item of information. A pub/sub system may also remove upstream dependencies for many-to-one exchanges where there are multiple publishers of the same item of information, as long as traceability is not required.

Traceability means that any changes in the system's behavior can be traced to the component or components responsible for the change. This is an essential feature of any reliable autonomous system. In a pub/sub system all changes in behavior can be reduced to information exchanges. Therefore, from an information exchange perspective, traceability means recursively identifying who the publisher is for an item of information. One easy way to ensure traceability is to enforce that each item of information has one and only one publisher (one writer rule). Given this traceability constraint, pub/sub systems have a problem when it is desirable to have multiple producers of the same item of information as would be the case if one had redundant sensors. If there are multiple publishers of the same item in the data share then a consumer has no easy way of tracing who was responsible for a change in the item (without adding an upstream dependency that is dynamically tracking the identity of the publisher). Consequently, each publisher must have a unique item in the data share. This means that each subscriber must have multiple subscriptions, one for each producer, which adds upstream dependencies to the subscriber. Subscribers must be changed every time a producer is added or removed.

2.2. Arbiter Decoupling

One way to remove this type of upstream dependency, when traceability is required, is to use a component we call an information arbiter. The arbiter is responsible for either switching or combining information or data flows from multiple publishers based on arbiter parameters that are also published items. The output of each arbiter is a single item in the data share. Downstream subscribers need only subscribe to the output of the arbiter. If another publisher of information used by the arbiter is added or removed the arbiter changes but nothing downstream. Each input to the arbiter is a distinct item in the data share and can be traced. Traceability comes from logging the parameters of the arbiter to know when a change in arbitration occurred. This happens much less often than changes to the inputs of the arbiter. Using arbiters is much simpler than dynamically tracking multiple publishers all overwriting the same item of information. This removal of upstream dependencies using arbiters is shown below. We call this feature of removing both upstream and downstream dependencies through the use of a pub/sub shared data store and arbiters, dependency decoupling.

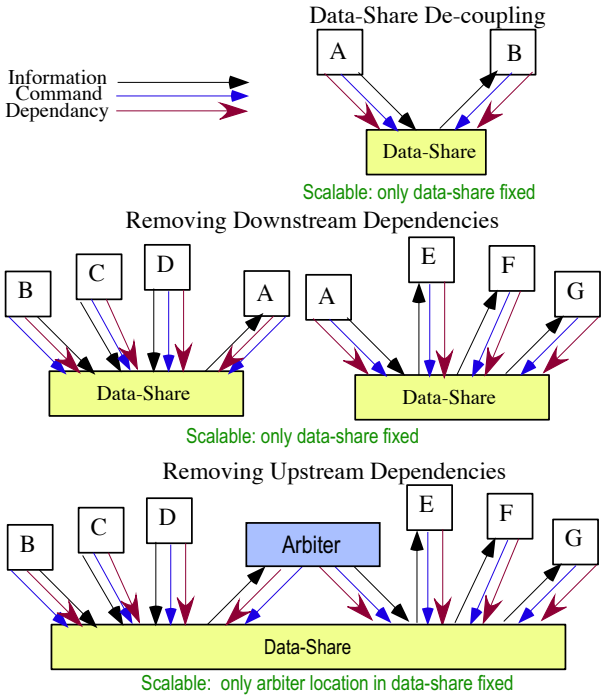


Fig.2.2: Down and Up Stream Dependency Decoupling via Shared Data Store and Arbiters

An example of an Arbiter component is shown below.

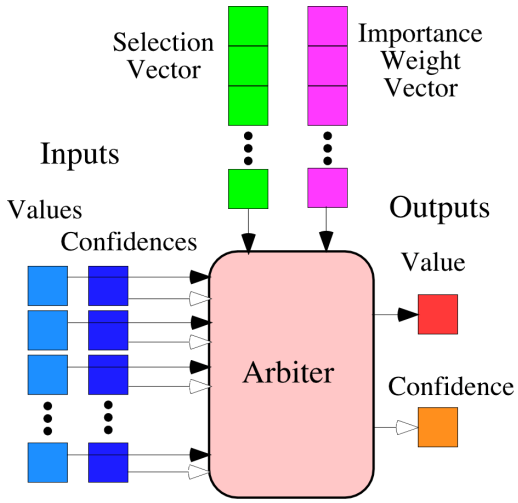


Fig.2.3: Information Arbiter

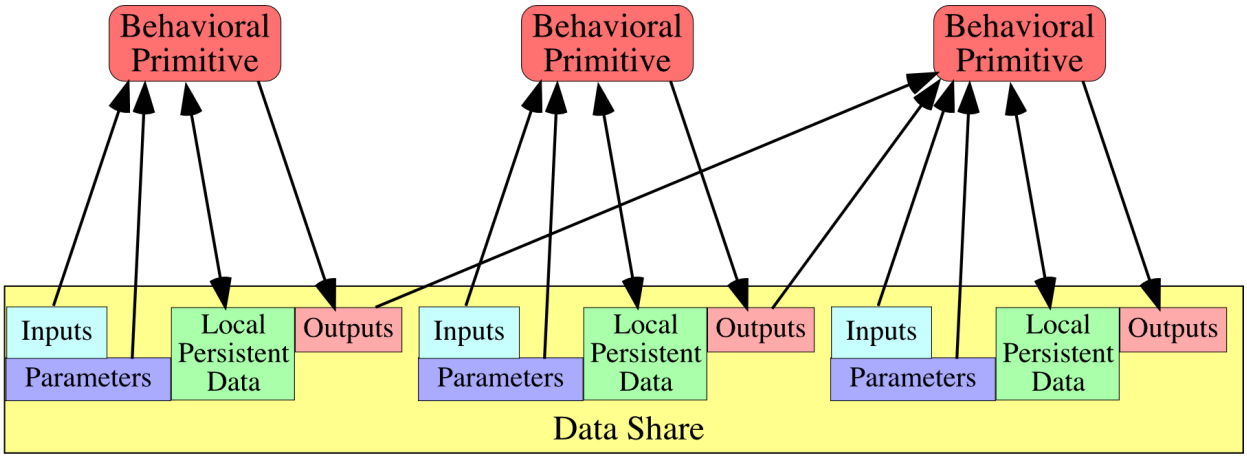


Fig.2.4: Interconnect of Components Through Data Share

A Shared Data Store does not have to be centralized. A distributed Store consists of nodes. Each node has a local copy of only the information that it needs. Each local Store is responsible for updating all other Node's copies of the same information. By default information in the Store is considered to be the best available at the time. In this case users of the data must be robust to the fact that data are not necessarily synchronized and coherent across the system. In cases where needed, and with extra effort, portions of the Store could be synchronized.

The Store provides the equivalent of a virtual distributed registry for information produced by other components of the system. The Store also provides a transparent interface to the external world. All information in the Store is observable by any subscriber. By storing the appropriate information in the Store the system becomes traceable. The Store provides certain vital book keeping functions such as time stamps and atomicity of structures and thereby provides a simple transparent way to provide remote monitoring and supervisory control. Resource management is abstracted into the much simpler concept of item ownership. The owner is the component authorized to change or write (publish) to a particular item in the Store. Transfers of ownership can be tracked and logged thereby providing traceability.

2.3. Other Techniques for Dependency Reduction

Another way of describing low apparent complexity is high transparency. A transparent system does not have a lot of details that block one's view of the system operation or that one must understand to understand the system. The amount of overhead and explicit details (dependencies) that must be kept track of by a component to exchange information with another is low. A transparent system also allows one to examine the critical dependencies or feature when needed.

Another way to reduce apparent complexity and perceived risk is by using late binding. In this context binding occurs when the information exchange between components becomes fixed. By late or early binding it is meant the point in the development and testing process when connections get fixed, such as, compile time, load time, install time, or run time. Usually, the later the binding the better.

Complexity is further reduced if the total number of different types of building blocks or components is small but can be combined or interconnected in a flexible manner. This is called power of expression.

2.4. Core Architecture

All of these techniques for dependency reduction described above have been employed in the core architecture that ioflo and floscript were designed to support. In summary, the core architecture of which ioflo is a component consists of reconfigurable component software modules that interface with a publish/subscribe shared data store and are scheduled by the HAF (hierarchical action framework) of ioflo. These software components allow convenient expression of control and planning algorithms as well as transparent monitoring, logging, and replay through the distributed publish/subscribe shared data store. The dependency reduction arising from modular components interfacing to a common store significantly reduces apparent complexity. The architecture infra-structure is unique in that it seamlessly unifies the scalable distributed data flow or port based component paradigm with the power and expressiveness of hierarchical discrete event systems. The architecture enables the formation of almost any semi-autonomous command and control system organization in a highly convenient manner.

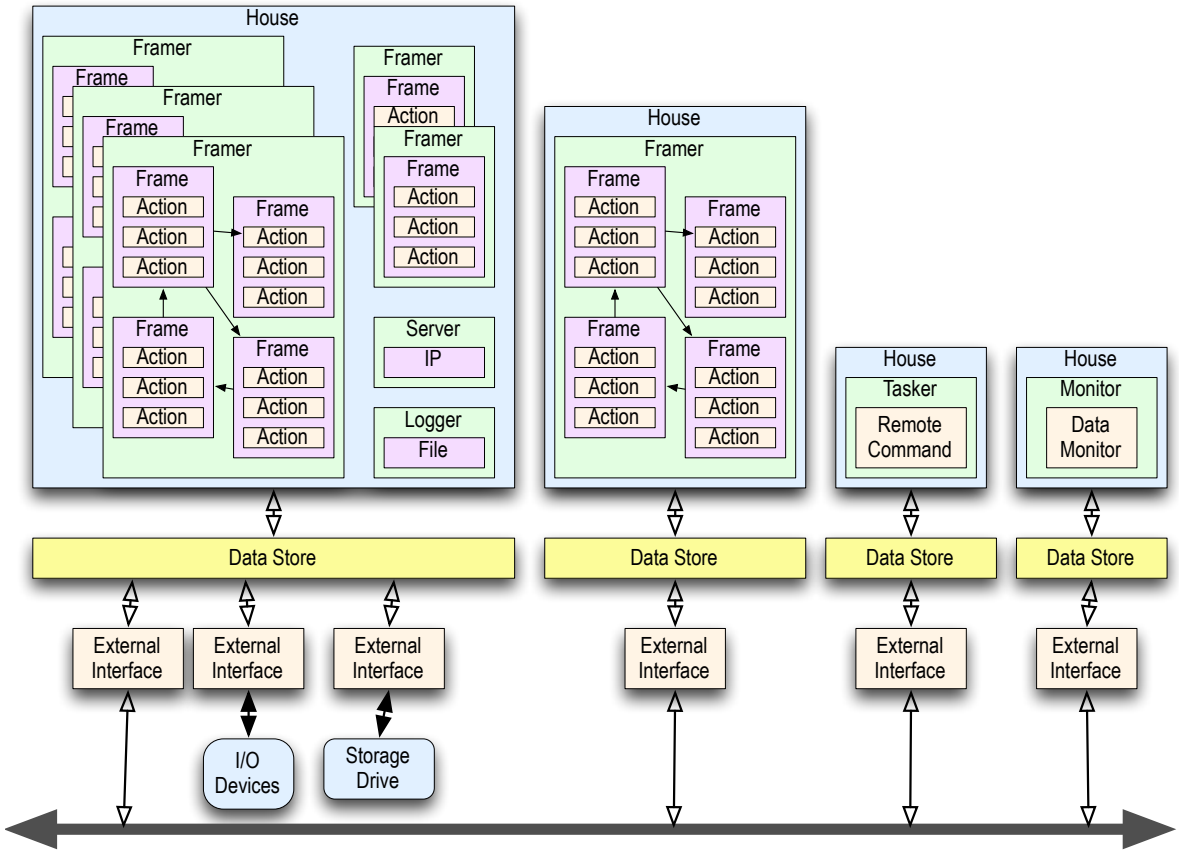


Fig.2.5: Core Component Based Pub/Sub Architecture

3. Hierarchical "State Machine"

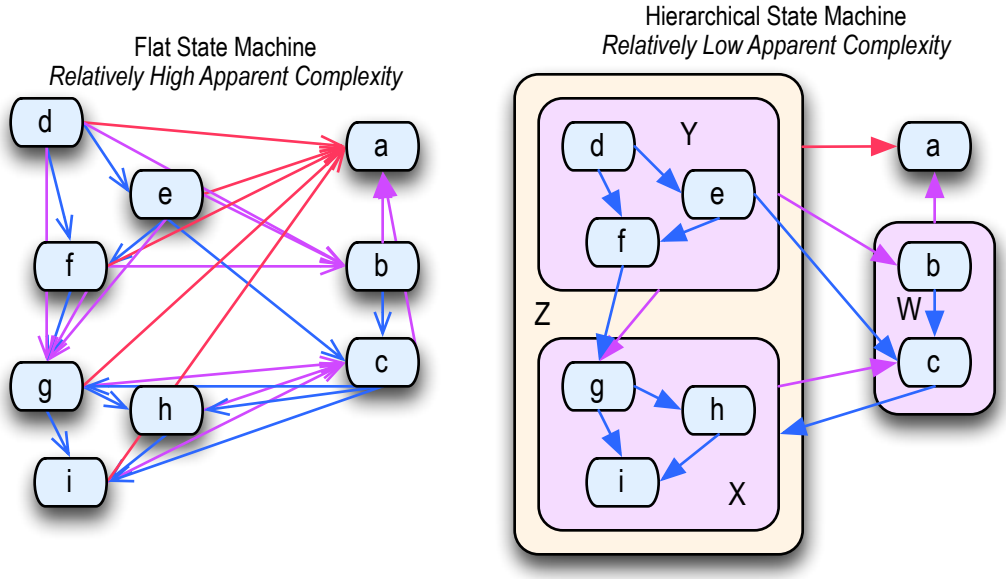
In addition to information exchange an autonomy architecture needs a way of configuring, organizing, scheduling, sequencing, and evaluating sensing, command, and control components.

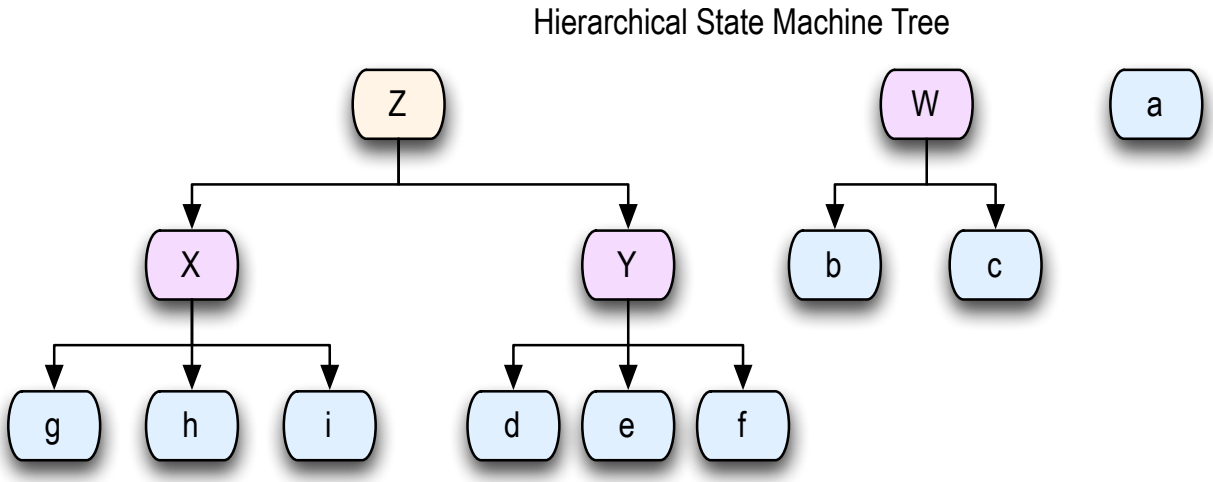
Each component action, agent, controller, etc in an autonomous command and control system using conventional computation equipment is at some level a form of state machine. We use the term "state machine" loosely here. By Hierarchical State Machine we mean a means of describing the evolution of a system through various states where those states are composed in a hierarchical manner. This is the sense used by StateCharts and UML (Unified Modeling Language). "State machines" provide a very convenient formalism for modeling component behavior. Moreover, autonomous control usually involves some sort of mission plan, that is expressed as a sequence of stages or activities, in other words a form of state machine as well. Various architectures use different levels of granularity to express this stages along a spectrum from a list of waypoints to general goal based behaviors like sweep and loiter.

Whether explicit or implicit, any autonomy software architecture running on a computer platform is a hierarchy of state machines. Moreover, human cognition is limited in how many distinct pieces of information can be thought of at one time. Thus it is difficult for a human to perceive simultaneously all the constituents of a complex system. Hierarchical composition/decomposition is one way, if not the only way, for humans to design and manage complex systems. More complex systems are built up from simpler subsystems where the mutual dependencies at each level are simple enough to be managed. Indeed it is the lack of a sufficiently general and flexible mechanism for hierarchical composition of the software components that eventually limits many autonomy architectures thus creating an apparent complexity barrier.

Since hierarchical composition is the "natural" way for humans to design complex systems, one would need a very compelling reason to use any other architecture. From a mission planning software perspective this means that a hierarchical state machine would appear to be the most appealing approach. Indeed, the Unified Modeling Language (UML) has at its core a hierarchical state machine modeling paradigm. We spent several years as part of a DoD funded research project comparing different high level command and control architectures with the conclusion that a *universal* command architecture could be expressed most conveniently using a hierarchical state machine formalism. Indeed hierarchical state machines are an excellent mechanism for dependency reduction.

Consider the following diagram. One the left side is a flat state machine, on the right side is an equivalent hierarchical state machine. Although the number of actual state transitions is the same the apparent number of state transitions in the hierarchical state machine is an order of magnitude less.





The actual evaluation of a hierarchical state machine, however, is not so simple. In a flat state machine, a state transition only needs to know about two states, the state being exited and the state being entered. In a hierarchical state machine a state is not simple. It is made up of several sub-states. We call them frames. A given state is then an ordered list of frames. A state transition involves resolving which frames are exited and which are entered from two ordered lists of frames. This is shown in the figure below.

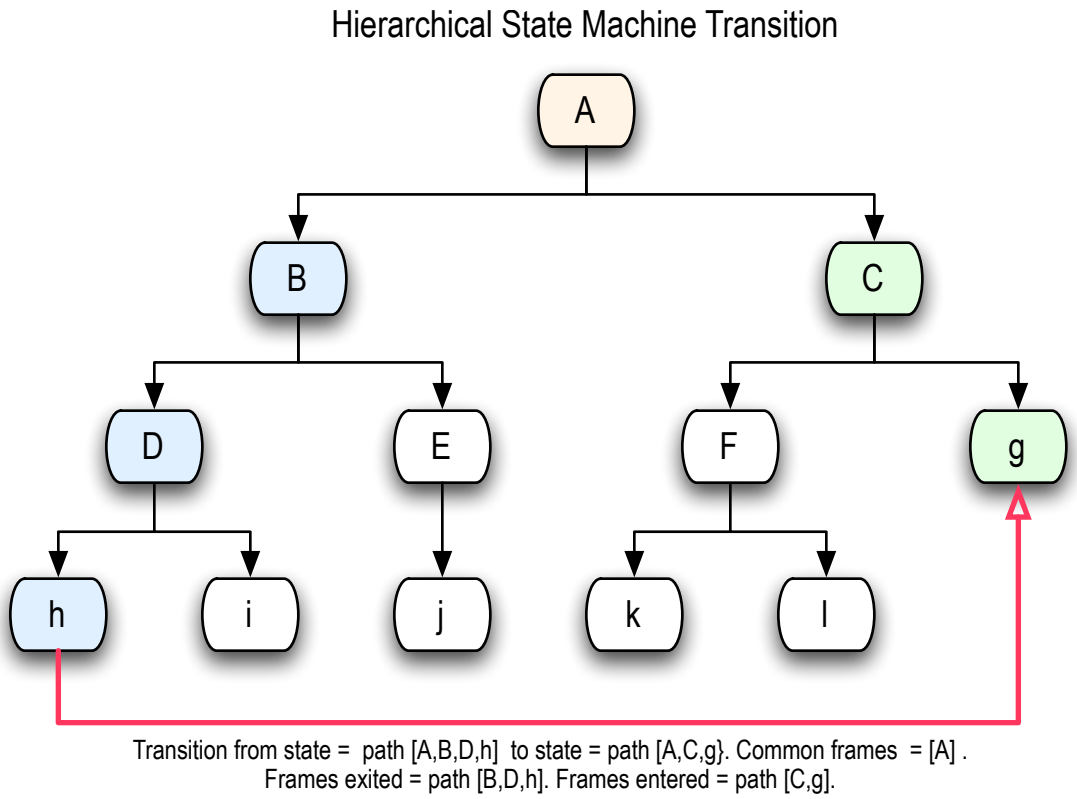


Fig.3.1: HSM Transition Paths

In order to use a hierarchical state machine (HSM) in a manner that truly reduces apparent complexity, there needs to be a convenient formalism for expressing and evaluating the HSM that hides the complexity of the transitions. If each HSM has to be hand crafted then there is still a complexity barrier. Indeed, the motivation for floscript is to provide such a convenient formalism.

Unmanned vehicle control systems must also employ "*safety jackets*", that is, failure and error handling routines that are often specific to a particular stage of a mission. A hierarchical state machine provides a uniquely convenient formalism for encoding these higher priority safety jackets without complicating the user programmed mission stages. We call this a *reliable services envelope*. The *reliable services envelope* is programmed by experts who understand the autonomous vehicles systems in detail. The reliable services envelope enables non experts to safely program the mission specific stages of behavior without an expertly detailed understanding of the vehicle systems.

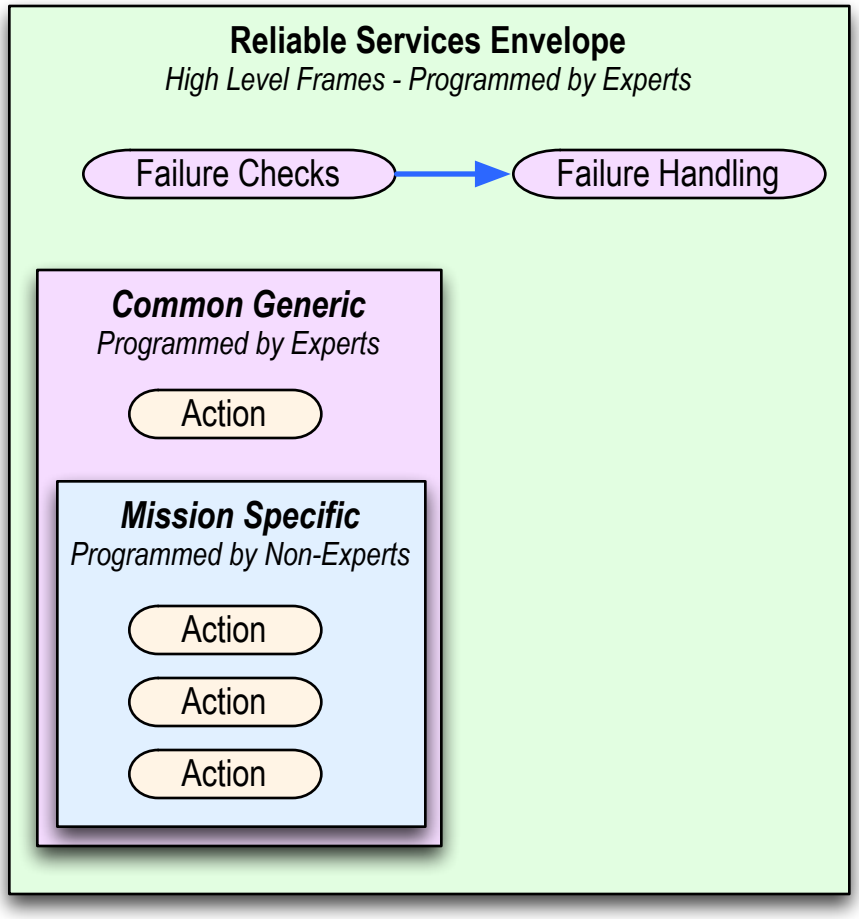


Fig.3.2: Reliable Services Envelope