The Convergence of Control, Communication, and Computation[*]

Scott Graham and P.R. Kumar

Department of Electrical and Computer Engineering, and
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL 61801
USA
{srgraham,prkumar}@uiuc.edu
http://black1.csl.uiuc.edu/~prkumar/

Abstract. The convergence of communication and computation over the past two decades has given us the Internet. We believe that the next phase of the information technology revolution will be the convergence of control, communication, and computation. This will provide the ability for large numbers of sensors, actuators, and computational units, all interconnected wirelessly or over wires, to interact with the physical environment. We argue that in the proliferation of this "convergence," a critical role will be played by the architecture. We describe an experimental Convergence Testbed at the University of Illinois, outline the architectural challenges, and our efforts in this direction.

1 Introduction

Over the past two decades we have seen the convergence of communication and computation, which has given us the Internet. Worldwide there are over 150 million internet hosts [1], and over 600 million users [2]. Indeed networked computers nowadays are critical not only for their computation capabilities but also for their communication capabilities. This phase of the information technology revolution has provided us the ability to exchange information in the form of email or to browse each other's webpages.

We anticipate that the next phase of the information technology revolution will provide us the ability to actively interact with the environment and alter it.

[*] This material is based upon work partially supported by USARO under Contract Nos. DAAD19-00-1-0466 and DAAD19-01010-465, DARPA under Contract Nos. N00014-01-1-0576 and F33615-01-C-1905, AFOSR under Contract No. F49620-02-1-0217, DARPA/AFOSR under Contract No. F49620-02-1-0325, and NSF under Contract No. NSF ANI 02-21357. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above agencies. Captain Graham is studying under U.S. Air Force sponsorship through the AFIT/CI program.

M. Conti et al. (Eds.): PWC 2003, LNCS 2775, pp. 458–475, 2003.

Such interaction will require sensing the environment and acting on it, and will be achieved by interconnecting sensors and actuators with computation elements, and providing all with communication capability.

Two technological trends making this feasible are the growth in embedded computers and wireless networking. About 98% of all microprocessors sold are embedded, and their percentage is growing [3]. They are present in cell-phones, watches, stereos, microwaves, washing machines, wireless thermometers, cordless phones and answering machines. They exist in pocket video games, VCRs, DVD players, printers, and components of computer systems. Automobiles typically have multiple such processors.

Currently these embedded devices function in an isolated way and are not significantly interconnected. The cost of wires alone is comparable to the cost of many of these devices, not to mention installation costs. Thus, toasters are not connected to alarm clocks. However we may be on the cusp of a wireless revolution. Wi-Fi (IEEE 802.11x) has experienced double-digit growth since 2000 [4], and is now installed as a default on several makes of computers. Lower cost wireless connectivity is possible with Bluetooth available at a $6 per chipset cost to manufacturers [5]. Extrapolating these trends in wireless communication, we can envision a time, not far off, in which wireless connectivity is a commodity. With each embedded device functioning as a sensor or an actuator, and each wirelessly connected with others, the future could well see orchestras of sensors and actuators playing over the ether in vast interconnected control systems. Indeed, the Berkeley Motes [6] already provide a combination of sensing, wireless communication, and computation, all in a package with a small spatial footprint and low energy usage.

In short we anticipate the convergence of control with communication and computation. But how will these systems interoperate? How will they be interconnected, physically, in applications, and in theoretical frameworks? These are the issues addressed in this paper. We argue that the architecture of these systems will play a critical role in their emergence and proliferation. We describe a testbed for convergence at the University of Illinois. We elaborate on the issues that arise, and outline an architectural solution being pursued by us to realize the twin goals of reliability and minimal design and deployment time.

2 Convergence Towards a More Holistic Theory

The aforementioned technological developments are leading to accompanying changes in research directions which are aimed at a more integrated view of systems theory. Though it may not be completely accurate to put too clear a historical marker, it can be said that the last half of the twentieth century was the age of developing the individual areas of control, communication, and computation. von Neumann's idea of a stored program (1944) and the ENIAC (1946) are about a half century old, and roughly mark the beginning of the age of computers. Wiener's second World War work, embodied in his "Yellow Peril" book (so known for the color of its cover and its perceived incomprehensibility) dates to 1949. Shannon's foundational information theory was published in 1948. Kalman's work on providing a foundation for state-space control theory dates to around 1960. In signal processing the seminal work of Cooley and Tukey is slightly more recent, around 1965.

However, we anticipate that the next few decades will witness the development of a more integrated system theory combining all these areas. For example, signal/image processing methods with information theoretic performance assessment and connections are already emerging [7,8]. Networking is seeing the confluence of computer science with more traditional communications research conducted in Electrical Engineering departments.(INFOCOM, for example, is jointly organized by the IEEE Computer and IEEE Communications Societies). Communication and control have a long history of involvement, dating back to the work of Wiener and Nyquist.

In the future, at the theoretical level (but not necessarily at the architectural level), issues such as addressing messages, and combining sensory inputs, while computing based on locally available data, will all be seen simply as tradeoffs in the context of design of a larger system.

As we enter this age of convergence, several challenges arise. In this paper we highlight one of them; how to design such "systems" (to use the EE term), or the "application" (to use the CS term), in an environment of constant change where new features are always being added, and hardware is constantly being changed, all without necessitating major recoding, and with a view to minimizing the designer's time. A closely related issue is what sort of software infrastructure (or "middleware" to use the CS term) is needed to facilitate the rapid development and deployment of systems.

Consider the IP stack in networking. It is present in all computers and has provided essential communication services by making interconnections transparent to the user. But what will the "IP stack" equivalent be for distributed and interconnected embedded systems? A suitable architectural construct will need to provide the appropriate services for sensors, actuators, computation, and communication to work together. Such infrastructure code should self-organize, taking care of details such as which computation is running on which host, and relieve the designer from mundane details such as IP addresses and the problem of start-ups, etc. The software should provide the right abstractions and interfaces to application programmers, and a rich set at that, so that they can concentrate on developing applications.

We can today build one-of-a-kind systems to fit just about any single use case, although such systems could be, and often are, enormously expensive. But that is not the vision we are projecting. Our vision is broader. The convergence of communication with computation is today ubiquitous because of several forces. We contend that similar forces are at play which will lead to the confluence of control with communication and computation also becoming ubiquitous. Moreover, the resulting systems will need to adapt to changing uses.

Our goal, thus, is to move from an era of carefully hand-crafted systems to mass production of interconnectable devices, with easy to configure interfaces,

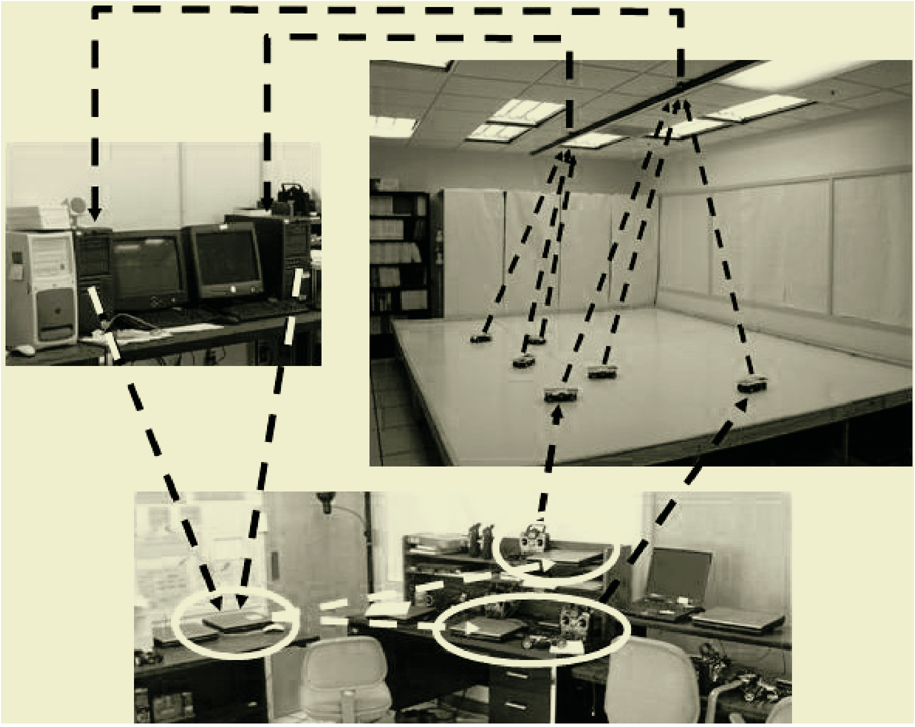

Fig. 1. Convergence Laboratory Testbed at the University of Illinois

such that systems which feature the convergence of control with communication and computation are routinely deployed with short design and development time, while incorporating flexibility to meet changing needs.

3 A Testbed: The IT Convergence Lab

To investigate these issues, we have set up an "IT Convergence Lab" at the University of Illinois, which features a testbed, as shown in Figure 1.

There are several reasons for the use of an experimental testbed. First, it represents a complete system, as we see below. Thus, while researchers often, and with good reason, focus in depth on a particular aspect of the overall system (say, the control law used, or the routing protocol, or the image processing algorithm), working with the system in its entirety educates us about all aspects of the system. Thus we are able to identify what is the critical bottleneck in the system at any given stage in the design. This may vary at different stages of evolution, as has been witnessed by us, between the search strategy used in predictive control, to the cycle time of computation in the image processing algorithm which needs to locate cars under non-uniform lighting, etc. Thus we are able to assess the impact of the choice of a particular strategy for a subsystem on the overall system's QoS. Additionally, the holistic aspect of the system serves a pedagogical role both for students and researchers. Instead of working in an abstract setting, the testbed forces us to be pragmatic in the context of a real system.

The particular choice of the testbed described below is useful in that it is a malleable system, which is at the same time simple. It allows us to investigate substantial aspects of middleware development as well as application development. Our goal is to develop and explore the principles important to the proliferation of control, communication, and computation, and to provide at least one concrete application and working implementation which incorporates these principles.

Our testbed consists of a fleet of fifteen cars. The cars are radio controlled (RC), which allows us to avoid mounting laptops on them, thus allowing us to keep the cars small in size, which in turn allows us to operate a number of them on a small indoor track, 12 feet wide by 16 feet long. The cars are controlled by off-board laptops, with each car having its own dedicated laptop. The serial port of each laptop is connected to a microcontroller, which in turn is connected to a transmitter which drives the particular RC car. The radio transmitters for the cars use separate non-interfering frequencies, and should essentially be regarded as dedicated "wires," emulating a scenario where the laptops are indeed mounted on the cars and connected to them by wires. The entire set of a car, its transceiver, its microcontroller, and its laptop, could just be considered as a single "car unit".

For communication between computers, the lab uses an ad-hoc wireless network, comprised of IEEE 802.11 PCMCIA cards, which carries the control and data packets used in the application. For monitoring and as a diagnostic backup, there is also a dedicated wired network consisting of an Ethernet switch and a small hub. The hub connects two VisionServers to a DataServer and the network switch.

The function of the vision system is to provide feedback to individual cars in the form of position and orientation information. Two overhead cameras continually monitor the platform on which the cars are mounted, and serve as the only sensors in the system. The video feed from each camera is sent over a cable to its dedicated desktop PC, called a VisionServer, where image processing is done to determine the orientations and locations of all the cars. To accomplish this, we have placed six color patches on each car's "roof," in distinct patterns. This color coding allows the vision system to distinguish cars. The vision system segments each of the pixels in a frame into predetermined colors, then searches for groups of colors, and identifies cars through their color patterns. When patterns are correct, the vision system then extracts the location of the color patches and determines the position and orientation of the car. The position information from the two desktops is sent over an Ethernet cable to a DataServer laptop, which in turn is connected to all the other laptops (controlling the cars) by the ad hoc wireless network. Figure 1 shows the complete "loop" on the testbed.

For simplicity and malleability, we employ dedicated laptops (all running Linux) for each radio controlled car. For now, the controller for each car runs on its dedicated laptop. However, as described in the paper [9] describing the software infrastructure and middleware aspects of this project, the next phase of our software infrastructure development will involve automatic migration of code so that the controller for a given car could be running on any of several laptops. For example, computations can move to locations which have maximum computational resource availability or require minimal communication resources. Or, to enhance reliability without sacrificing efficiency, multiple controllers may run on one laptop with other laptops standing by in case of controller failures.

Thus the entire system features multiple sensors (currently two vision systems, with more that can be added as necessary), multiple actuators (the fifteen cars), and multiple computational resources, with loops closed over an ad hoc wireless network.

We refer to systems of this type as Federated Control Systems [9]. In principle, we should be able to replace the cars with airplanes, vision systems with GPS or other sensors, and have an air traffic control system. The architecture should be the same. Or, should we replace the VisionServers with thermometers, door sensors, motion detectors, and smoke detectors, and the cars with heating controllers, sprinkling systems, lighting controllers, etc, the resulting system should be an easily re-configurable total home control system.

But before such systems are common, they must be inexpensive, easy to use, and useful.

4 The Importance of Proliferation

Proliferation is important for proliferation. There is positive feedback. Individual one-of-a-kind systems may serve to fill a particular need. But when these systems are mass-produced, they become inexpensive, and the demand for their use increases. This in turn leads to improvements, which further increases demand, driving down cost, and so on. Many of the eventual uses may be of limited value, and hence would never support large scale costs of development on their own, but when the costs come down and are amortized over a huge number of applications, these lower value needs will begin to drive the market.

Our goal is to address system design challenges arising when we move towards this level of demand in the vision of convergence. We contend that the critical ingredient involved in realizing this goal is architecture.

5 Importance of Architecture

Ultimately, the usefulness of an overarching design depends on how well it adapts to the particular needs of an individual context, while at the same time capturing the essence of that which is common across all the usages. The particular useful variations of a product may not be known to the designers beforehand.

We believe that successful proliferation will depend on providing the right abstractions and architecture for use by designers. We begin by providing some examples of technologies which have successfully proliferated.

We contend, and this may be controversial, that the success of the Internet is primarily architectural, and secondarily algorithmic, though protocols such as TCP have played a most important role. Consider the OSI model of networks, even though in practice it is not followed precisely. This model separates the various functions of communication into layers of abstractions, giving a specific purpose to each layer and hopefully enabling it to perform at that layer. (There are many cross-layer design issues as well that deserve attention; see [10]). Services at a layer can be oblivious to lower layers, and hence can focus on that portion of the design which has been delegated to them. In order to interoperate, we must, of course, provide interfaces between layers which are well-defined and understood by both sides of the interface. With well-defined interfaces above and below each layer in the protocol stack, it becomes possible to make changes to an intermediate layer without affecting the other layers in the system. This allows for incremental evolution. It gives the design longevity.

Consider the alternative of merging all the layers so the implementation runs faster. Such a system would not have longevity since any small change would necessitate the redesign of the entire system. It would simply not be conducive to proliferation.

In addition to longevity, the overall scheme must provide a rich enough set of abstractions to support individual uses. It must provide an architecture that allows a designer to visualize where different aspects of a problem are solved and where they are located.

Indeed, architecture is important for proliferation of technologies, in general. Valiant [11] claims that the success of serial computation is due to the von Neumann bridge, and contends that it is the lack of a von Neumann bridge that is one of the reasons for the failure of certain efforts in parallel computation.

In communication, we contend that the separation of source coding from channel coding has played a major role in the proliferation of digital communication. Indeed we argue that this structural result established by Shannon [12] has been more important than the precise characterization of channel capacity.

Similarly, more important than the precise values of the feedback gains is the closed-loop architecture of a control system. The very separation of the overall system into a portion that cannot be modified, the “plant,” and a portion that can be, the “controller,” is important, and is obvious only in retrospect. Indeed, simulation software often does not make this distinction, with the result that implementing certain policies may require digging into code where the plant and controller interactions are intermingled.

These examples illustrate the fundamental and far reaching influence of architecture on systems and motivate the desire to address the architecture appropriate for the convergence of control, communication, and computation.

6 Application Architecture for Ever Evolving Systems

We now turn to the issue of concern in the paper – how to design a large system featuring a multiplicity of sensors and actuators.

Our approach is motivated by several fundamental considerations. The overarching issues are:

(i) The ever changing nature of a system,
(ii) The complexity of the design.

7 The Need for the Incremental Evolution of the Design of Complex Systems

The first consideration is that the design of a large system is always in flux. It is never at an end. As a system is built, new features are always added.

In the early mass production of WWII aircraft, US automobile manufacturers assumed that automobile assembly-line methods would translate to aircraft manufacturing, without a strong understanding of the additional complexity of aircraft and the manufacturing precision required. Frequently, design changes were required even before the first aircraft would come off the line. Rather than change the assembly line, the fixes were often done in separate modification centers. Even then, further changes were often made at front-line bases [13].

Similarly, it is erroneous to design today's large and complex systems under the assumption that software is easy to change, and therefore adaptable. The ability of a system to adapt to changing requirements depends heavily on the overall architecture of the system and the nature of the changes. Only if the system is well-designed, with flexible architecture, can one hope that the resulting system will be adaptable.

An important driver of change is "feature bloat," though we do not use the phrase in a pejorative sense. But it must be carefully managed. One starts with a modest goal, and an eye toward future changes, and completes it reliably. Then one inserts additional functionality to make the system more useful. Indeed, this is an ever present feature of many software projects. (Successive versions of Microsoft Word are just one prominent example).

Similarly, viewed from the usage end, customers do not always know what they want or need at the beginning of a design cycle. Upon experiencing a new capability, they may envision slight variations that would make the capability more useful. Apparently small changes can, however, have large unintended negative effects as they ripple through the design of a complex system. Systems should thus be well-designed a priori, to the extent possible, so as to be able to incorporate this inevitable feature bloat, and insulate the risk of feature failure from other parts of the system which must be reliable.

Incremental development may also be necessary from an economic point of view in the proliferation and mass adoption of a technology. A system under development for an extended period of time will not produce any financial support for the developer during the development phase. Thus, for large development efforts, it is useful to build the system in smaller increments, each of which provides an increase in functionality. This produces continuous revenue, making the proliferation phase financially viable.

Incremental development also provides useful feedback in the design and application of the system. As increments are tested, identified problems can be resolved before future increments suffer from the need for redesign. We can see this principle in the early development of our testbed. In the beginning, we simply worked to get a single car running in open loop, according to a preplanned sequence of speed and steering commands. In this phase, the cars were found to be too slow, and the motors unreliable. We did not need to have an entire system working to discover this. Moreover, this discovery led to changes in the motors and gearboxes which would have changed all of the calibration data for each car. We had not yet invested time calibrating every car, thus early feedback helped to avoid this time consuming task for the remaining cars. We were also initially concerned about slack in the steering mechanism and hence the repeatability of the cars performance. Several open-loop demonstrations proved that the cars were sufficiently repeatable to meet our needs, thereby avoiding a redesign of the steering which we had thought necessary.

Of course, incremental upgrades must be relatively simple to incorporate at each stage. Moreover, it is useful to be able to "roll back" if an upgrade fails in some fashion. This ability to "undo" is a challenge to system design, but provides much needed flexibility to designers and users alike.

8 Design Goals for Application Architecture

Thus, we contend that complex system design must be regarded as a continuing process. Our purpose is to address the issue of design of such a system so as to meet two goals:

(i) Reliability.
(ii) Minimizing the time to design and deploy a new feature.

The need for reliability is now well accepted. Indeed, rather than "high performance," the focus of much of current software research is on "reliability." It may be referred to as robustness, or fault-tolerance, or perhaps security as well, highlighting particular aspects of system reliability. In any case, reliability is now entrenched as a primary performance criterion.

The second focus is on cost—in terms of human time. It is important to reduce both the time to design a system, as well as the time to deploy it. In fact, in our approach these go hand in hand. By abstracting certain aspects of systems in appropriate ways, we aim to realize our ultimate vision that these systems be "mass produced" rather than "hand crafted." Our process aims to reduce the process to one of designing an individual block and then easing its implementation through a process of selecting among interoperable components and setting appropriate constraints. Thus we aim to make one-of-a-kind systems affordable and hence useful.

Our design process views the overall system as a composition of "decision-making modules," segregating levels of decision making to broadly conform to commonly used and well understood methods for control system design, and adopting an evolutionary approach that we call "Collation." This provides support for reliability and evolution at many architectural layers.

9 The Levels and Modules of Decision Making

At the highest architectural level of any application is the system goal, which may change over time. A clean architectural design separates goals at all levels of decision making from the means to accomplish them.

High level goals are further instantiated or translated as they percolate through the layers of the design to elements of the system that are more aware of information that pertains to the optimization of the goals. One common refinement of goals is by time scale decomposition [14], though there are also other possibilities such as, for example, spatial decomposition.

As one moves across layers, one finds perhaps several steps of control refinement and data abstraction. As a concrete illustration, an air traffic controller need not know the precise settings of the throttle on a particular aircraft, which may however be of utmost interest to the pilot. Similarly, the pilot may be unaware of aircraft movement on the ground at the destination airport, even though such movement will ultimately affect the pilot's operating conditions. The air traffic controller need only know the plane's position, airspeed, and flight plan in order to accomplish the higher level goal of ensuring flight safety, while the pilot need only know if the flight plan is still acceptable to the controller, and need not know about the flight plans of any other plane. Of course, a fail safe mechanism is in place locally. Upon detecting an oncoming plane, a pilot will change course for safety without clearing it with the air traffic controller who probably does not have data at the level of refinement the pilot needs, or the ability to make a decision fast enough to help.

From decision theory [15], we can regard any decision making unit as subject to three "inputs." First is the "goal," which could, for example, be specified as a cost function to be optimized. Second is the "model." This provides the basis for deciding what will be the result of actions, and thus allows us to choose between different actions when seeking to optimize the cost function. Third is the "information" available to the decision making unit. This could be noisy measurements of say the locations of the cars, etc. The output of the decision making module can be regarded as the optimal (or near optimal, or satisfactory) choice of an action which does well by the goal, based on the model of the environment, and the available measurements.

To illustrate these issues more concretely, in the Convergence Lab Testbed we have a centralized top level planner called the ScheduleServer which is responsible for generating collision-free timed trajectories for each car along routes

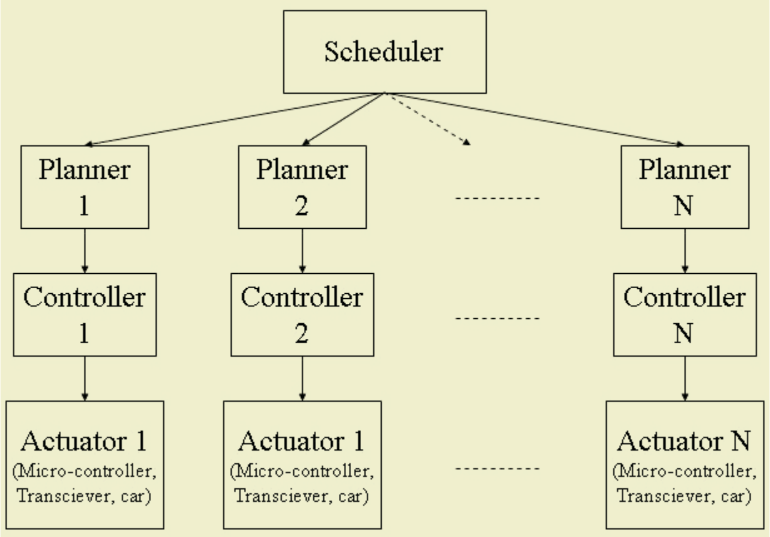
Scheduler
Planner 1
Planner 2
Planner N
Controller 1
Controller 2
Controller N
Actuator 1 (Micro-controller, Transciever, car)
Actuator 1 (Micro-controller, Transciever, car)
Actuator N (Micro-controller, Transciever, car)

Fig. 2. Goal decomposition in the testbed

which represent the high level goals of the system [16]. The goals may be specified merely as a triple comprising an origination location, an intermediate way-point, and a final destination on the track. The server must then be able to determine feasible paths which satisfy the constraints of the track, and then schedule collision-free trajectories for each car.

The output of this plan is a set of timed way-points for each car. This is given to the middle level in the task architecture, which is distributed. Specifically, each car has it's own mid-level controller called the Planner.

The ScheduleServer monitors the locations of the cars and determines when something has gone wrong and re-plans, sending new trajectories which replace the old ones. Figure 2 shows the goal decomposition/refinement on the testbed.

The Planner in normal mode merely passes the trajectory on to the low level real-time controller which dutifully attempts to track the trajectory. However, the Planner has also been given access to position and orientation information for each of the other cars, or obstacles, on the track. By monitoring their movement, the Planner can perform additional functions such as following another car in a formation or in a pursuit-evasion scenario. It may also provide collision avoidance by predicting potential collisions, which may occur even with collision-free trajectories due to misbehaving cars, malicious cars, or simply dead batteries. Upon detection of imminent collision, the Planner may simply stop the car to prevent the collision, or instead plan an alternative path based upon some criterion. This recovery planning then represents a mid-level goal. It is not the high level goal, but represents refinement of a higher level goal as a result of additional information. Moreover, in the face of unpredicted behavior, it is able to prevent system failure.

The low level real-time controller is myopic. In our case, we have traded cheap computational power for algorithmic complexity. The real-time controller uses a linearized model of the car and examines thousands of potential control sequences, comparing the predicted results of each of them with the desired result, choosing the one with the lowest cost, i.e., the one which most closely follows the desired trajectory. Note that at this level, the goal is merely to conform to the given trajectory. There is no understanding of higher goals such as collision avoidance. Separation of concerns in this hierarchical structure greatly simplifies the design, while providing flexibility at the same time.

It should be mentioned that there is really one more lower layer. The microcontroller which sends signals to the car stores a sequence of controls, which for fail-safe reasons is always terminated with a stop command. Failure to receive an update from the low level real-time controller is thus handled appropriately in the hardware/firmware level of the system. Such bottom end failsafe mechanisms are necessary in safety critical applications and could even be implemented through run-time invariance checking in micro-processors.

Using a proper task architecture, we have separated responsibility among the various parts of the system. Incorporating new tasks is greatly simplified. For example, if one of the cars represents an ambulance, and we want it to have priority, then the only part of the system which is affected is the very top level ScheduleServer which must be altered to give priority to it.

10 Reliability and Dependence

As noted earlier, the first attribute of “performance” may well be reliability, which in turn has many dimensions.

In any system, certain portions of the system may depend on other parts for its operation. A cellular phone clearly “depends” critically on its antenna to complete a call. However, some dependencies are not inherently necessary, but creep in as design or implementation dependencies[1]. For example, a laptop may only “use” wall power to operate but does not “depend” on it since it has a battery. Complex systems exhibit complex dependencies which are difficult to trace, and pose challenging troubleshooting problems. Many design or implementation dependencies could perhaps be converted into “use” relationships. Consider the power steering system in a car. If the power steering fails, the power assist to the driver is gone, but the steering wheel remains physically connected to the movement of the front wheels, allowing a driver to continue to drive safely, although requiring increased effort.

To understand dependencies, it is useful to visualize the space of errors and categorize them. We will enumerate a few for purposes of this discussion.

Execution. This includes all forms of system crashes, all segmentation faults, all power failures, deadlocks, livelocks, infinite loops, and any other faults which

[1] We are grateful to Professor Lui Sha for educating us about what the requirements for reliability are in practice, and how to address them.

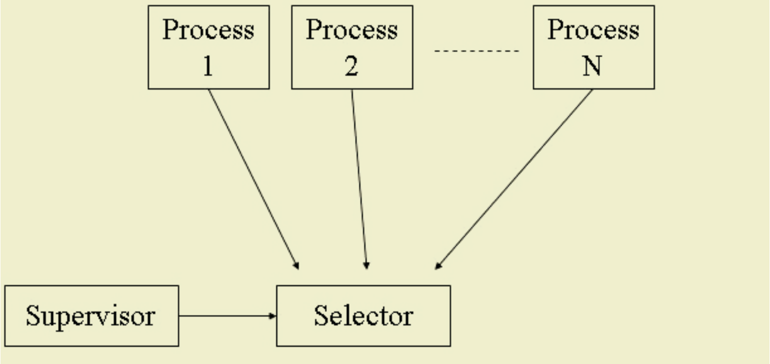

Fig. 3. Collation architecture

prevent a process from executing. There is a great deal of research on the prevention of execution errors. We are interested in how to continue to operate in the presence of such errors.

Timing. This includes any operation which does not return the result within the deadline required. It is important in systems which interact with their environment.

Semantic. This means that while the function proceeded and returned a result by the deadline, its value was not correct in some sense. This could simply be a design flaw, and is frequently ascribed to the application or domain expert.

11 Collation for Evolution

In keeping with traditional principles of functional programming, our design is as modular as we are able to achieve. Thus the focus is on code reuse rather than rewrite. When adding new features, our goal is to "insert" functionality rather than revamp the existing architecture.

The ability to "undo" an action while word processing a document provides tremendous flexibility. Similarly, instead of being critically dependent on correct operation of a more complex implementation, we can merely use it when it is satisfactory, and revert to a simpler version when it is not. It is therefore reasonable to deduce that large complex systems must incorporate the ability to switch between components when they fail or when increased functionality is desirable, while maintaining system integrity in the face of faults, failures, and changes in operational environments. Such capability must be built in, and even itself upgradeable perhaps. Figure 3 presents the architectural construct, or design pattern, of "Collation."

The Convergence Testbed has experienced many system failures and system upgrades as well as changes in the operational environment. As these changes have occurred, we have observed the utility of a process of "Collation" as a design pattern and have incorporated it wherever feasible. An example concerning the

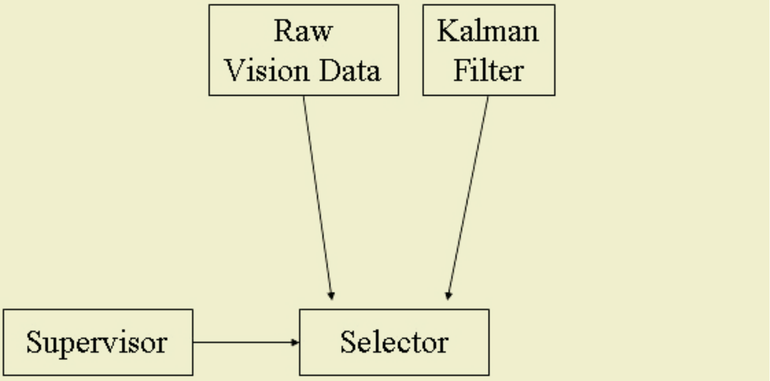

Fig. 4. Kalman filter inserted with Collation

incorporation of a vision data filter illustrates this method. In an early version of the testbed, with just one car running, and just one camera, the vision system was not responsible for identifying the car, but just reporting its position and orientation. A reliable system for this functionality was in place, and the real-time controller was able to use the raw vision data reliably for its operation. Moreover, there was no need for a centralized store of vision data; therefore, the DataServer was not yet implemented. The system performed its task of following predetermined trajectories quite well in this early version. However, at a later stage, to improve the smoothness of trajectory following, it was decided to add a Kalman filter. This was done by adding it as a parallel block to an existing direct breakthrough. During the debugging phase, the existing position and orientation information which was "reliable" but not "very accurate" was used to monitor the Kalman filter's output. Figure 4 illustrates the Collation process applied to the Kalman filter.

This idea of Collation can be cast into the Simplex architecture of [17]. Sha, *et al.* [18] have considered the use of a simple reliable controller as a backup to a complex, unreliable controller. This method is based on using simplicity to control complexity. The key notion is that the simple controller, previously established to be reliable in some way, can always maintain stability of the system and meet certain safety parameters provided that the system state is within a well defined operating region, as for example, the basin of attraction of its Lyapunov function. Then a smaller region is defined within which the complex controller is given authority over the system. A supervisory process, which must also be reliable, observes the system state in order to determine if and when the complex controller will cause the system state to move outside the basin of stability of the simple controller. When this occurs, the supervisor switches control to the simple controller, thereby maintaining stability.

Collation encompasses more than redundant safety systems. It extends to what we call "data fusion." Consider multiple data sensors in a system. An aircraft avionics system may receive position information from GPS, land based beacons, and inertial navigation system, as well as manual updates from a navigator. This data can be fused in several ways. Perhaps the data is averaged. This works well if the data all have similar noise characteristics. But if one of the sources were providing grossly erroneous data, averaging is not the best fusion. Perhaps the system can compare all of the inputs and vote on which sources are reliable. If multiple sources agree, then they are trusted and used. Of course, as the fusion becomes more sophisticated, the likelihood of introducing errors grows. So the Collation process provides the simple algorithms to run alongside the complex versions just in case.

Another usage of Collation lies in assessing the effects of time-delays in the incremental deployment of a more complex control system. Control systems are generally sensitive to timing. Delays introduced into a stable control loop can even render it unstable. A system designed with the ability to switch between a stable version of a process and an experimental version, can accommodate online development and testing safely. Consider, as a simple example, a filter located somewhere along a control feedback loop. When a more sophisticated filter is being entertained, the additional processing required for it may introduce additional delay, which could render it worse than the original simpler design. By applying the Collation design pattern, we are able to first program another version of the simple filter that includes the additional delay, without any algorithmic changes, and use the Collation process to switch between the original and the delayed versions, monitoring the system for undesirable effects. Once we have tested this sufficiently, we may then install the full functionality of the complex filter and run it in place of the delayed version of the original filter. Because of the supervisor, we can make these changes at run-time (in real-time) without bringing down the system. Moreover, the original filter is still in place, ready to be used in the event of undesirable behavior of the complex filter. So Collation facilitates incremental operational testing by allowing low risk online upgrade.

Yet another place where Collation is useful is in "planning." Multiple plans can be generated and evaluated, and the plan with the best performance can be implemented. One example of this in the Testbed is in the mid-level Planner which continuously monitors the vision data, predicting where cars will be in the next several steps and comparing the current trajectory with those positions in order to predict future collisions. Upon detection of a potential collision, the Planner may create several alternative plans, perhaps a path to the left of the collision, and perhaps one to the right. These alternate paths are then checked for collisions, and if one is deemed successful, it is used. If not, the desired behavior is to come to a stop, and the Planner accordingly stops the low level controller, thereby avoiding a potential collision. Figure 5 shows Planning inserted via Collation.

These examples illustrate the fundamental ability to connect to, and select among, multiple sources of data or control. Properly implemented, this functionality provides for evolution, rollback or undo, and reliability. It provides a separation of decision criteria, or rules, from the execution of the criteria. We

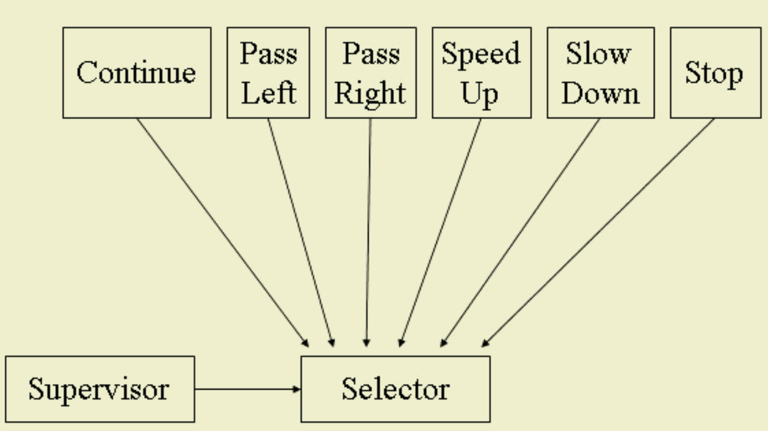

Fig. 5. Planning as Collation

can implement implementing the Selector as a separate process from the other components, we can create the Simplex Architecture for reliable on-line system upgrade. By choosing among many pieces of source code for compilation, the process can be called software configuration management. When multiple external components can connect to a single component (which can in turn provide its output to multiple external components) and each component has local "intelligence" contained in the Supervisor to govern its action, we can realize an interoperable Federated Control System. If such "intelligence" includes rules for fail-safe operation, then the system has fail-safe at that level of abstraction.

Collation bears a resemblance to object oriented design. In aggregation or hierarchical composition, each element of the Collation architecture can be a trivial one, or a very complex system of its own, or something in between. Thus, Collation exhibits a self-similar nature useful for hierarchical construction and decomposition.

The design pattern that we call Collation combines widely used principles into a useful architectural construct. Whether or not a system realizes the benefits of connection to and selection among multiple sources depends upon the software design of the underlying infrastructure.

A movie showing several applications of the functioning system is available on the Testbed website [19].

12 Concluding Remarks

The architect Christopher Alexander argues in [20] that in any system underlying patterns of use exist. The principles, or forces, which drive the use of that system must be understood, in order to design it in such a way that it comes alive, which is to say that it fulfills the purpose of its creation. Many systems may experience similar kinds of forces, and patterns of design then emerge which can be seen throughout many similar structures.

We believe that identification of such “design patterns”[21], incorporating them into the architecture, providing the infrastructure that allows routine deployment of converged systems, and providing the designer with a rich set of abstractions, is critical to realizing the convergence of control with communication and computation.

References

1. “Internet Domain Survey,” Internet Software Consortium, Jan 2003. http://www.isc.org/ds/WWW-200301/index.html.
2. “How Many Online?”, Nua Internet Surveys, 2003. http://www.nua.ie/surveys/how_many_online/.
3. J. Stankovic, “VEST: A Toolset For Constructing and Analyzing Component Based Operating Systems for Embedded and Real-Time Systems,” University of Virginia TRCS-2000-19, July 2000.
4. K. Carter, A. Lahjouji, N. McNeil, “Unlicensed and Unshackled: A Joint OSP-OET White Paper on Unlicensed Devices and Their Regulatory Issues,” May 2003, http://hraunfoss.fcc.gov/edocs_public/attachmatch/DOC-234741A1.doc.
5. V. Lipset, “In-Car Bluetooth To Grow Beyond Telephony, Study Says” May 23, 2003 http://www.thinkmobile.com/Everything/News/00/67/32/.
6. “Wireless Sensor Networks,” CrossBow Technology Inc, http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.
7. S. S. Pradhan and K. Ramchandran, “Distributed source coding using syndromes (DISCUS): Design and construction,” *Proceedings of the IEEE Data Compression Conference (DCC)*, Snowbird, Utah March 1999.
8. R. Shukla, P. L. Dragotti, M. N. Do, M. Vetterli, “Rate-distortion optimized tree structured compression algorithms for piecewise smooth images (448 kB),” *IEEE Transactions on Image Processing*, Jan. 2003, submitted.
9. G. Baliga and P. R. Kumar, “Middleware Architecture for Federated Control Systems,” IEEE Distributed Systems Online, June 2003, http://dsonline.computer.org/0306/f/bal.htm.
10. V. Kawadia, P.R. Kumar, “A Cautionary Perspective on Cross Layer Design,” Technical Report, CSL, University of Illinois, Jun 28, 2003. http://black1.csl.uiuc.edu/∼prkumar/ps_files/Cross_Layer.ps.
11. L. G. Valiant, “A Bridging Model for Parallel Computation.” *Communications of the ACM*, vol. 33, no. 8, August 1990.
12. C. E. Shannon, “A Mathematical Theory of Communications,” Bell System Technical Journal, vol. 27, pp. 379–423 and 623–656, July and October, 1948 (2 parts).
13. J. Rumerman, “The American Aerospace Industry During World War II,” *US Centennial of Flight Essay*, 2003, http://www.centennialofflight.gov/essay/Aerospace/WWII_Industry/Aero7.htm.
14. S. B. Gershwin, *Manufacturing Systems Engineering*, Prentice-Hall, Englewood Cliffs, NJ 1994.
15. D. Blackwell and M. Girschick, *Theory of games and statistical decisions*, Wiley, New York, NY, 1954.

16. A. Giridhar and P. R. Kumar, “Scheduling Traffic on a Network of Roads,” Technical Report, CSL, University of Illinois, Apr 2003.
http://black1.csl.uiuc.edu/∼prkumar/ps_files/trafficpaper.ps.
17. L. Sha, R. Rajkumar, and M. Gagliardi, “The Simplex Architecture: An Approach To Building Evolving Industrial Computing Systems,” *Proceedings of the International Conference on Reliability and Quality in Design*, pp. 122–126, Seattle, Washington, Anaheim, CA, ISSAT Press, March 16–18, 1994.
18. L. Sha, R. Rajkumar, and M. Gagliardi, “Evolving Dependable Real Time Systems,” *Proceedings of IEEE Aerospace Conference*, Vol. 1, pp. 335–346, Aspen, Colorado, IEEE Computer Society Press, February 3–10, 1996.
19. “The Convergence Laboratory testbed,” University of Illinois,
http://black1.csl.uiuc.edu/∼prkumar/testbed/.
20. C. Alexander, *The Timeless Way of Building*, Oxford University Press, Oxford, UK, 1979.
21. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1995.