Using Simplicity to Control Complexity

Lui Sha, University of Illinois at Urbana-Champaign
lrs@cs.uiuc.edu

Key Words: software fault tolerance, process diversity, forward recovery, complexity, simplicity, reliability, and automatic control.

Abstract: How to improve the reliability and availability of the increasingly complex software is a serious challenge as software assumes an increasingly larger role in the critical functions of our society. It is a widely held belief that diversity in software constructions entails robustness. However, is it really true? This paper investigates the relationship between software complexity, reliability, and the resource available for software development. It also presents a forward recovery approach based on the idea of "using simplicity to control complexity" as a way to improve the robustness of complex software systems.

1 Introduction

"As our economy and society become increasingly dependent on information technology, we must be able to design information systems that are more secure, reliable and dependable."[1] There are two basic approaches for software reliability. One is the fault avoidance method using formal specification-verification methods [1] and a rigorous software development process such as FAA's DO 178B standard for flight control software certification. These are powerful methods that allow us to have computer controlled safety critical systems such as flight control. Unfortunately, they can only handle modestly complex software. The trend towards using large networked system of systems based on commercial-off-the-shelf (COTS) components also makes the application of fault avoidance method more difficult.

Another approach is software fault tolerance using diversity. It is a widely held belief that diversity entails robustness. However, is it really true? Would the system be more reliable if we devote all the efforts to developing a single version than dividing the efforts for diversity? In this paper, we show that, depending upon the architecture, dividing the resource for diversity could lead to either improved or reduced reliability. The key to improving reliability is not the degree of diversity per se. Rather, it is the existence of a simple and reliable core component that can ensure the critical properties of the system, in spite of the faults of the complex software. We call this forward recovery approach "using simplicity to control complexity". In this paper, we show how to use this approach systematically in the domain of automatic control applications.

In Section 2, we analyze the relationship between diversity, complexity, reliability and development effort. In Section 3, we present the idea of "using simplicity to control complexity" and show that it can be systematically applied to software for automatic control applications. Finally, Section 4 is the summary and conclusion.

[1] Information Technology for 21st Century: A Bold Investment in America's Future, http://www .ccic.gov/it2/initiative.pdf.

2 The Power of Simplicity

All software projects have finite budgets. Will we have a more reliable software system by putting all the efforts into a single program, or by dividing the limited resource for diversity?
To get some insights into this question, let's develop a simple model to analyze the relationship between reliability, development effort and the logical complexity of software. Computational complexity is modeled as the number of steps to complete the computation. Likewise, logical complexity can be viewed as the number of steps that are needed to verify the correctness. It is a function of the number of cases (states) that must be handled in the verification or testing process. A program can have different logical and computational complexities. For example, comparing with heap-sort, bubble-sort has lower logical complexity but higher computational complexity. This paper focuses on logical complexity.

Another important point we want to make is the distinction between logical complexity and residual logical complexity. A program could have high logical complexity initially. However, if it has been verified and can be used as is[2], then the residual complexity is zero. However, software reuse is not easy and most of the time, modification or extension is needed. Residual complexity models the effort that is needed to ensure the reliability of this modified software. For a new module, residual logical complexity and logical complexity is the same. Residual complexity is low if the modification is expected by the original design. If the modification is incompatible with the architecture of the original design, it could be harder to verify than to write a new one.

In this paper, we focus on residual logical complexity because it is a dominant factor in software reliability. In the following, the term "complexity" refers to residual logical complexity unless stated otherwise. From a developer's perspective, the higher the complexity, the harder to specify, design, to develop, and to verify. From a management perspective, the higher the complexity, the harder to precisely understand the users' needs and to communicate them to developers, to find effective tools, to get qualified personnel and to keep the development process smooth without a lot of requirement changes. Based on what has been observed in software development, we make three postulates:

- *P1: Complexity Breeds Bugs:* Everything else being equal, the more complex the software project is, the harder to make it reliable.
- *P2: All Bugs Are Not Equal*: The obvious errors are spotted and corrected early during the development. As time passes by, the remaining errors are subtler, more difficult to detect and correct.
- *P3: All Budgets are Finite:* There is only a finite amount of effort (budget) that we can spend on any project.

P1 implies that for a given mission duration t, the reliability of software decreases as complexity increases. P2 implies that for a given degree of complexity, the reliability function has a monotonically decreasing rate of improvement with respect to development effort. P3 implies that diversity is not free. That is, if we go for diversity, we must divide the available effort in some way.

We now develop a simple reliability model that satisfies these three postulates. For a normalized mission duration $t = 1$, the reliability function of a software system can be modeled as an exponential function of the software complexity, C, and available development effort, E, in the form of $R(E, C) = e^{-kC/E}$, where k is a scaling constant. For simplicity, we shall assume that $k = 1$ in the

[2] It is important to point out that we cannot simply use a known reliable component in a new environment without verifying the assumptions made by the component.

rest of this paper. Figure 1 is a plot of the reliability function $R(E, C) = e^{-C/E}$ with $C = 1$ and $C = 2$ respectively. As we can see in Figure 1, the higher the complexity, the more effort is needed to achieve a given degree of reliability. $R(E,C)$ also has a monotonically decreasing rate of reliability improvement, modeling the fact that as time passes by, the remaining errors are subtler, more difficult to detect and correct.

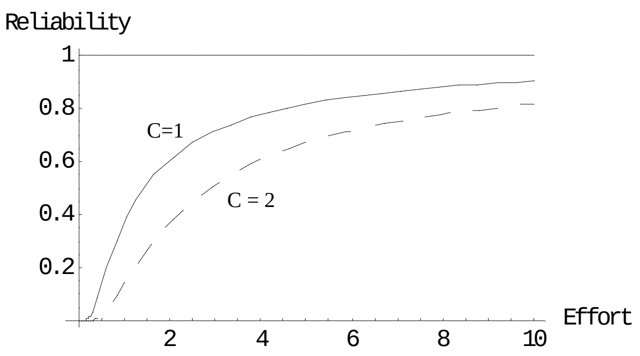

Figure 1: Reliability and Complexity

Another way to arrive at the same formulation is to adopt the commonly used exponential reliability function $R(t) = e^{-\lambda t}$ and assume that the failure rate, λ, is proportional to the complexity of the software, C, and inversely proportional to the development effort, E. That is, $R(t) = e^{-kC.t/E}$. Again, for simplicity, we shall assume that the scaling constant $k = 1$. To focus on the interplay between complexity and development effort, we normalize the mission duration t to 1 and write the reliability function with a normalized mission duration in the form of $R(E, C) = e^{-C./E}$.

We now have a simple model that allows us to analyze the relationship between development effort, complexity, diversity and reliability. The two well-known software fault tolerance methods that use diversity are N-version Programming and Recovery Block[2, 3, 4]. We shall use them as examples to illustrate the application of this simple model.

To make a fair comparison, we shall make the comparison under each method's preferred ideal condition. That is, faults are assumed to be independent under N-version Programming and perfect acceptance test is assumed to exist under Recovery Block. Neither of these two assumptions is, however, easy to realize in practice. This leads to the forward recovery approach [6]. We shall address this subject later in this paper.

To focus on the effect of dividing the development effort for diversity, we assume that the nominal complexity is $C = 1$. Let's first consider the case of N-version Programming with $N = 3$. The key idea of N-version Programming is to independently design and implement different versions of programs from the same specification, hoping that faults resulted from software errors could be independent. During runtime, the results from different versions are voted on and the majority is selected[3].

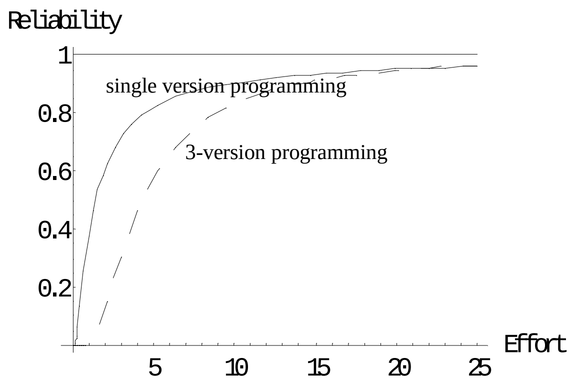

Figure 2.1: Effect of Divided Efforts in 3-version Programming, When the Failure Rate is Inversely Proportional to Effort

In the case $N = 3$, the reliability function of the 3-version Programming system is $R_3 = R_{E/3}^3 + 3R_{E/3}^2(1 - R_{E/3})$. The

[3] The median is used if outputs are floating point numbers.

reliability function of each version $R_{E/3}$ is obtained by replacing E with $(E/3)$ in $R(E, 1) = e^{-1/E}$, since the total effort E is divided by 3 teams. Each team is responsible for a version. Figure 2.1 shows the reliability of a single version programming with undivided effort is superior to 3-version Programming over a wide range of available development effort.

This result counters to the common belief that diversity entails reliability. To check the sensitivity of this result, we make two different assumptions. First, we make a more optimistic assumption that the failure rate is inversely proportional to the square of software engineering effort, i.e. $R(E,1) = e^{-1/E^2}$. Second, we make a more pessimistic assumption that the failure rate is inversely proportional to the square root of software engineering efforts, $R(E,1) = e^{-1/E^{1/2}}$. Figures 2.2 and 2.3 are plots under these two assumptions respectively. As we can see, the reliability of a single version is better than that of the 3-version Programming under these two assumptions over a wide range of efforts.

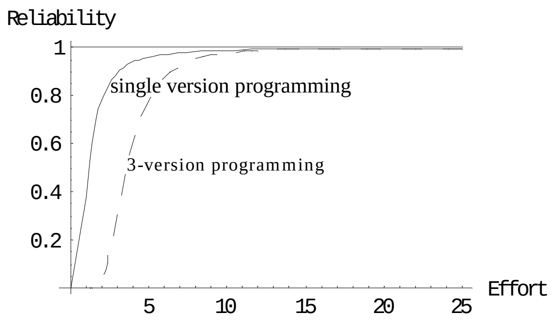

Figure 2.2: Effect of Divided Efforts in 3-version Programming, when failure rate is inversely proportional to the square of effort

As we can see, dividing the development efforts for diversity does not guarantee the improvement of reliability. In fact, it could be counter-productive, even under the assumption of fault independence.

However, we should not conclude that single version programming is always superior to N-version Programming. Sometimes, additional versions can be obtained inexpensively. For example, if one were to use POSIX operating systems, one could easily add a new low cost version by using the Linux operating system. To model the reliability precisely, in addition to the cost consideration, we also need to be careful about the fault independence assumption and model the effect of correlated faults.

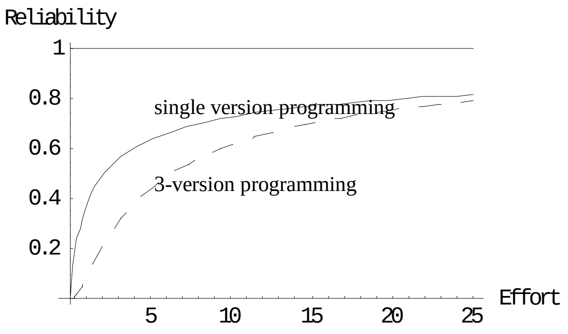

Figure 2.3: Effect of Divided Efforts in 3-version Programming, When the Failure Rate is Inversely Proportional to the Square Root of Effort

We now consider the effect of diversity in the context of Recovery Block. The basic idea of Recovery Block is to construct different alternatives and subject them to a common acceptance test. When input data arrive, the system checkpoints its state and then executes the primary alternative. If the result passes the acceptance test, it will be used. Otherwise, the system rolls back to the check-pointed state, and tries the other alternatives until either there is an alternative that passes the test, or the system raises an exception that it cannot produce a correct output.

When there is a perfect acceptance test, the system works as long as any one of the alternatives works. When there are 3 alternatives, the reliability of the Recovery Block system is $R_B = 1 - (1 - R_{E/3})^3$, where $R_{E/3} = e^{-1/(E/3)}$. Figure 3 shows the reliability of a single version programming and the reliability of the Recovery Block with a three way divided efforts. When the available effort is low, single version programming is better. However, Recovery Block quickly becomes better after $E > 2.6$.

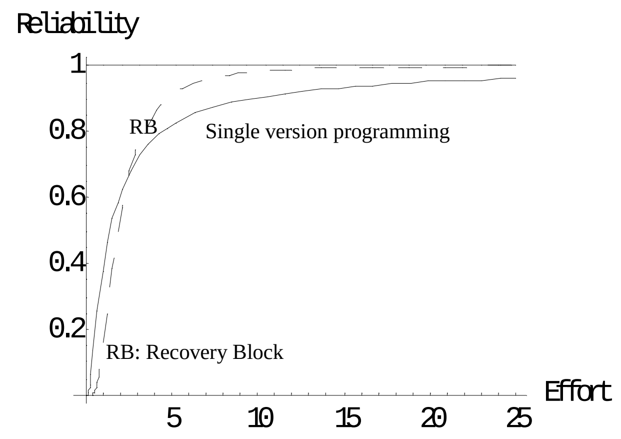

Figure 3: Effect of Divided Efforts in Recovery Block

A moment's reflection tells us that Recovery Block scores better than N-version Programming, because under Recovery Block, only one version needs to be correct while N-version Programming requires that the majority of the versions is correct. It is easier to have one out of N versions correct than to have the majority of N versions correct.

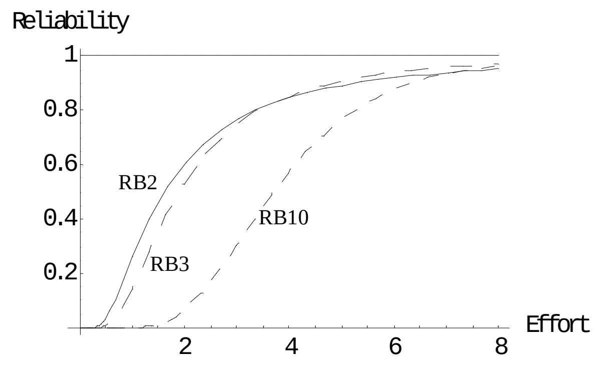

Figure 4: Degree of Diversity

Diversity in the form of Recovery Block helps. But is it the more diversity the better? Figure 4 compares the system reliability under Recovery Block when the total effort E is divided evenly into 2 alternatives (RB2), 3 alternatives (RB3) and 10 alternatives (RB10) respectively. All the alternatives have the same nominal complexity $C = 1$. As we can see, dividing the available efforts in many ways is counter-productive.

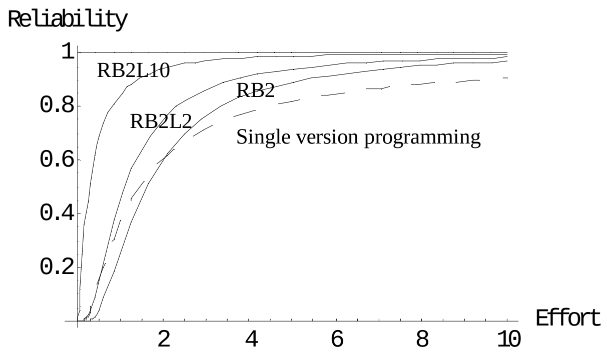

Figure 5.1: Effect of Complexity Reduction

Next, let's consider the effect of using a reduced complexity alternative. Figure 5.1 shows the effect of reduced complexity alternative in a two alternative Recovery Block. In this plot, the total effect E is divided equally into two alternatives. RB2 has two alternatives with no complexity reduction, i.e., $C_1 = 1$ and $C_2 = 1$. RB2L2 has two alternatives with $C_1 = 1$ and $C_2 = 0.5$. RB2L10 has 2 alternatives with $C_1 = 1$ and $C_2 = 0.1$. As we can see, the system reliability improves significantly when one alternative is significantly simpler. Indeed, Recovery Block recommends the use of a simpler alternative.

To underscore the power of simplicity, let's consider the effect of a good but imperfect acceptance test. Suppose that if the acceptance test fails the system fails. Figure 5.2 plots three reliability functions: 1) single version programming, 2) Recovery Block RB2 with *perfect* acceptance test and two alternatives, where each has complexity $C = 1$, and 3) Recovery Block *RB2L5 with imperfect acceptance test whose reliability is equal to that of the low complexity alternative, and two alternatives $C_1 = 1$ and $C_2 = 0.2$. As we can see, with a 5-fold reduction of complexity, *RB2L5 is superior to both single version programming and Recovery Block with perfect acceptance test but without the reduction of complexity in its alternatives.

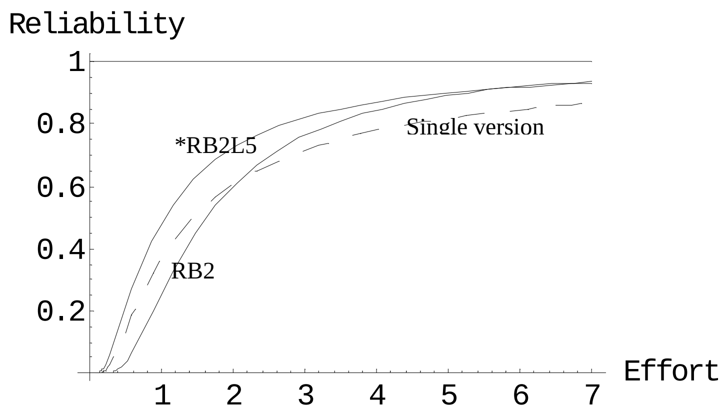

Figure 5.2: Effect of Complexity Reduction under Imperfect Acceptance Test

The key to improve reliability is not a high degree of diversity per se. Rather, it is the existence of a simple and reliable core component that can be used to ensure the critical properties. This observation is not a surprising result. After all, "keep it simple" has been the "mantra" of reliability engineering. What is surprising is that it has not been taken seriously in the construction of software systems as they assume an increasingly larger role in the critical functions of our society

A two alternative Recovery Block with a reduced complexity alternative is an excellent approach whenever high reliability acceptance tests can be constructed. Unfortunately, it is often difficult to construct effective acceptance tests to check the correctness of each individual output in many practical situations. For example, from a single output, it is impossible to determine if a uniform random number generator is indeed generating random numbers uniformly. The distribution becomes apparent only after a large number of outputs become available. Many phenomena share this characteristic: it is difficult to diagnose them with an individual sample but become evident when a large sample is available. Unfortunately, many computer applications are interactive in nature. They do not permit us to buffer a long sequence of output and analyze them first before using them.

Fortunately, we can leverage the power of simplicity using forward recovery.

3 Using Simplicity to Control Complexity

In this section, we will show how to systematically implement the idea of "using simplicity to control complexity" in the domain of automatic control. We begin this section with a discussion of the basic concepts and intuitions.

3.1 A Conceptual Framework

The wisdom of the reliability engineering commandment, "keep it simple", is self-evident, and has been repeated by countless generations of engineers in all the engineering disciplines. We all know that simplicity is the pathway to reliability. The importance of simplicity has also been well analyzed and documented by Leveson in [5]. Why is it so difficult to keep systems simple?

One reason is the pursuit of features and performance. To gain a higher level of performance and functionality than the state of practice permits requires us to push the technology envelope, and to stretch the limits of our understanding. Given the keen competition on features, functionality and performance in the market places, the production and usage of complex software components, either in the form of custom components or COTS components, are unavoidable in most applications. Useful but not essential features cause most of the complexity.

From a software engineering perspective, the notion of using simplicity to control complexity allows us to separate critical properties from desirable properties. In addition, this approach allows us to leverage the power of formal methods. To illustrate this point, let's consider the problem of sorting. In sorting, the critical property is to sort items correctly. The desirable property is to sort them fast. Suppose that we can formally verify the Bubble Sort program but unable to verify the Quick Sort program. Can we make use of this slow Bubble Sort as a watchdog for Quick-Sort? Yes, we can.

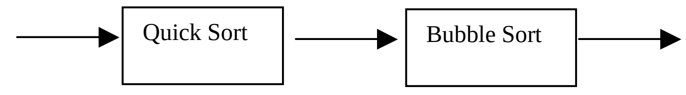

Figure 6: Using Simplicity to Control Complexity

As illustrate in Figure 6, if Quick-Sort works correctly, the items are already sorted when they are sent to Bubble Sort. Hence, the expected computational complexity is still *O(n log(n))*. If items were sorted incorrectly, Bubble Sort will sort them correctly and thus guarantee the critical property of sorting at lower performance. Under this arrangement, we can not only guarantee the correctness of sorting but also have a high performance as long as Quick Sort works most of the time. The moral of this story is that we can exploit the features and performance of complex software even if we cannot verify them, as long as we can guarantee the critical properties by simple software. This is how we leverage the power of formal methods and high assurance development process. Finally, we note that this is not an isolated example. Similar arrangements could be made for many optimization programs that have logically simple greedy algorithms with lower performance, and have logically complex algorithms with higher performance.

We would like to develop a systematic procedure to handle the software fault tolerance in real-time control applications. First, software now controls the operation of most modern artifacts, from small cell phones to cars, airplanes and factories. Indeed, the much-anticipated smart cars, smart buildings and smart cities are about the intelligent control of networked devices. Second, control applications often have high reliability and/or availability requirements.

Feedback control is a form of forward recovery. What the feedback loop does is to continuously correct errors in the plant (device) state. We need feedback, because there is neither perfect mathematical model of the plant nor perfect sensors and actuators. There is always a difference (error) between the actual plant state and the set-point (desired state). In the framework of feedback control, incorrect control software outputs translate to actuation errors. What we must do is to limit the loss resulted from incorrect outputs, and to keep the system in a state within operational constraints. One way to achieve this goal is to constrain the system states under the control of complex components within an envelope established by a simple and reliable controller. This is the technical meaning of "using simplicity to control complexity". We will discuss this in detail in the next section.

A noteworthy example of "using simplicity to control complexity" in practice is the flight control system of Boeing 777[8]. It uses triple-triple redundancy for system level reliability. At the software application level, it uses two controllers. The sophisticated control software specifically developed for Boeing 777 is the normal controller. The secondary controller is based on the control laws originally developed for Boeing 747. The normal controller is much more complex and is able to deliver optimized flight control over a wide range of conditions. On the other hand, control laws developed for Boeing 747 have been used for over 25 years. It is a matured old technology – simple, reliable and well understood. From our perspective, we will call it a simple component since it has low residual complexity. To exploit the advantage of advanced control technologies and to ensure a very high degree of reliability, Boeing 777 under the normal controller should fly within the stability envelope of its secondary controller. This is a fine example of forward recovery for potential faults in complex software systems.

However, the use of forward recovery in software systems is an exception rather than the rule. Forward recovery also receives relatively little attention in software fault tolerance literature [7] due to the perceived difficulties. Indeed, for a long time, how to systematically design and implement a forward recovery approach in feedback control has been a difficult problem without a general solution approach. Using the recent advancement in the theory of the Linear Matrix Inequality (LMI) [11], we have found that the forward recovery problem for automatic control system can be designed and implemented systematically as long as the system is piecewise linearizable. This covers most of the practical control applications. We have implemented this approach in a dynamic real-time architecture known as the Simplex Architecture [9]. Simplex architecture and its extensions have been successfully applied to a semi-conductor wafer making facility [10] and to F16 flight control using Lockheed Martin's F16 flight simulator[4]. We also developed a web based control laboratory, the Telelab at www-drii.cs.uiuc.edu/download. Telelab uses a physical inverted pendulum that can be controlled by *your* software to illustrate the principles discussed in this paper. Your software, once submitted, will be hot-swapped with the existing control software on-the-fly and take control of the inverted pendulum. You can watch how well your software improves the control of the inverted pendulum via streaming video. You can also test the reliability of this approach by embedding *arbitrary* bugs in your control software. In this case, our system will detect the increase of control errors, switch off your software, and keep the pendulum from falling down.

3.2 The Simplex Architecture

First, let's consider the development of the high assurance control subsystem. Complexity is a leading cause of software errors. To ensure the reliability, we follow the reliability-engineering adage – "keep it simple". This high assurance control subsystem is also the place where the power of formal methods and rigorous development methods can be effectively used and then leveraged to ensure the integrity of the overall system. The Simplex architecture consists of two subsystems: the high assurance and high performance control subsystems.

The high assurance control subsystem

- Application level: using well-understood classical controllers designed to maximize the envelope of vehicle stability with a reduced level of control performance to keep the control software simple.

[4] This work was done under the INSERT project, sponsored by DARPA and US Air Force Research Laboratory. More information on this project can be found in www.cs.cmu.edu/Groups/real-time/insert.

- System software level: using high assurance OS kernels such as certifiable Ada runtime developed for Boeing 777. This is a "no thrill" runtime that avoids complex data structure and dynamic resource allocation methods. It trades off usability for reliability.
- Hardware level: using well-established and simple fault tolerant hardware configurations, such as pair-pair or TMR.
- System development and maintenance process: using a suitable high assurance process appropriate for the applications, for example, FAA DO 178B for flight control software.
- Requirement management: requirements here are limited to critical properties and essential services. Like the constitution of a nation, they should be stable and change very slowly.

This conservative high assurance control core is then complemented by a high performance control subsystem. In safety critical applications, the high performance subsystem can utilize more complex advanced control technology. The same standard of rigor must also be applied to the high performance control software. The prototypical example here is Boeing 777's two controllers.

On the other hand, many industrial control applications, e.g., semi-conductor manufacturing, are not safety critical but the down time can be very costly. With a high assurance control in place to ensure the process remain to be operational, we can aggressively pursue advanced control technologies and cost reduction in the high performance subsystem as described below.

The high performance control subsystem

- Application level: using advanced control technologies, including those that are difficult to verify, e.g., neural nets.
- System software level: using COTS real time operating systems and middleware that are designed to simplify the development of applications. To facilitate the upgrade of application software components, we can also add dynamic real time component replacement capability in the middleware layer, which allows us to replace software components during runtime without shutting down the systems [9].
- Hardware level: using standard industrial hardware, such as VME or industrial PCs.
- System development and maintenance process: using standard industrial software development processes.
- Requirement management: requirements for features and performance are handled here. With the protection offered by the high assurance subsystem, they can be changed relatively fast to embrace new technologies and to support new user needs.

Figure 7 is a block diagram that illustrates the Simplex architecture, which supports the use of simplicity to control complexity. The high assurance and high performance systems are running in parallel. The high performance software is also isolated from the high assurance control software. Normally, the plant is under the control of the complex software. The decision logic makes sure that 1) the instantaneous error under the control of high performance is

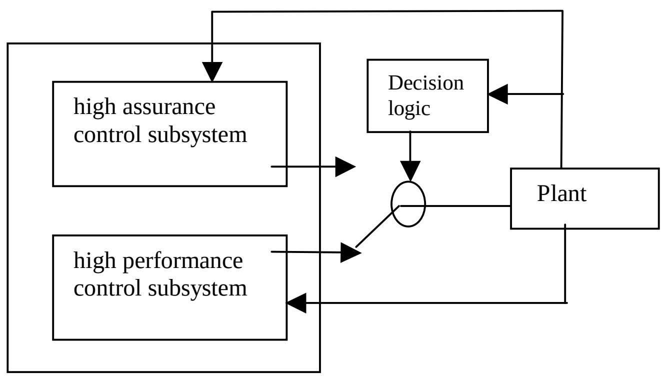
Decision logic
high assurance control subsystem
Plant
high performance control subsystem

Figure 7: The Simplex Architecture

not too large for the high assurance controller to handle, and 2) the average errors under the high performance controller should be smaller than that of the high assurance controller. Otherwise, the control is switched to the high assurance system.

3.3 Forward Recovery Using High Assurance Controller and Recovery Region

In this section, we review the development of the recovery region (also known as safety region) that ensures the plant will not violate the operational constraints. A detailed treatment on this subject can be found in [12]. In the operation of a plant (or a vehicle), there is a set of state constraints, called operation constraints, representing the safety, device physical limitations, environmental and other operation requirements. The operation constraints can be represented as a normalized polytope in the N-dimensional state space of the system as shown in Figure 8. Each line on the boundary represents a constraint. For example, the rotation of the engine must be no greater than *k* RPM. The states inside the polytope are called admissible states, because they obey the operational constraints. To limit the loss that can be caused by a faulty controller, we must ensure that the system states are always admissible. This means that we must be able to 1) take the control away from a faulty control subsystem and give it to the high assurance control subsystem before the system state becomes inadmissible, 2) the system is controllable by the high assurance control subsystem after the switch, and 3) the future trajectory of the system state after the switch will stay within the set of admissible states. Note that we cannot use the boundary of the polytope as the switching rule, just as we cannot stop the car without collision when the car is about to touch the wall. Physical systems have inertia.

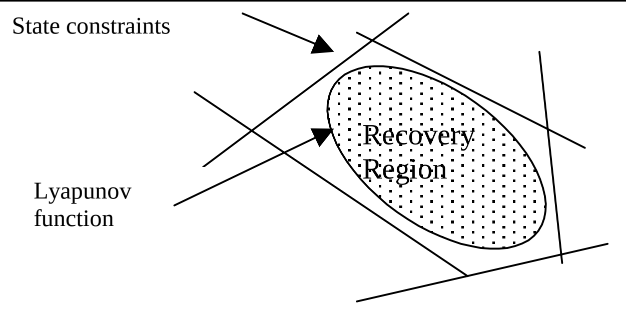

Figure 8: State Constraints and the Switching Rule (Lyapunov Function)

A subset of the admissible states that satisfies these three conditions is called a recovery region. The recovery region is represented by a Lyapunov function inside the state constraint polytope. That is, the recovery region is a stability region within the state constraint polytope. Geometrically, a Lyapunov function defines a N-dimensional ellipsoid in the N-dimensional system state space as illustrated in Figure 8. A Lyapunov function has the following important property. If the system state is inside the ellipsoid associated with a controller, the system states will stay within the ellipsoid and converge to the equilibrium position. Thus, we can use the boundary of the ellipsoid associated with the high assurance controller as the switching rule.

Lyapunov function is not unique for a given system and controller combination. In order not to unduly restrict the state space that can be used by high performance controllers, we need to find the largest ellipsoid within the polytope that represents the operational constraints. Mathematically, finding the largest ellipsoid inside a polytope can now be solved by the Linear Matrix Inequality (LMI) method [11]. Thus, we can use Lyapunov theory and the LMI tools[5] to solve our recovery region problem. For example, given a dynamic system $X' = A^* X + B K X$, where X is the system state, A^* is the system equation and K represents a controller. We can first choose K

[5] The software package that we used to find the largest ellipsoid was developed by Steven Boyd's group at Stanford.

by using well understood controller designs with a robust stability, i.e., the system stability should be insensitive to model uncertainty.

The system under the control of this reliable controller is $X' = A X$, where $A = (A^* + B K)$, where the stability condition is represented by $A^T Q + Q A < 0$, and Q is the Lyapunov function. The operational constraints are represented by a normalized polytope. The largest ellipsoid inside the polytope can be found by minimizing $(\log \det Q^{-1})$ [11], subject to stability condition. The resulting Q defines the largest normalized ellipsoid $X^TQX = 1$, the recovery region, inside the polytope as shown in Figure 7.

In practice, we use a smaller ellipsoid, e.g., $X^TQX = 0.7$, inside $X^TQX = 1$. The distance between $X^TQX = 1$ and $X^TQX = 0.7$ is the margin reserved to guard against measurement errors. During runtime, the plant is normally under the control of the high performance control subsystem. The plant state X is being checked at every sampling period. If X is within the N-dimensional ellipsoid $X^TQX = c, 0 < c < 1$, then the instantaneous error is considered acceptable. Otherwise, the high assurance control subsystem takes over the control. This ensures that the operation of the plant never violates the operational constraints. Finally, we note the software that implements the decision rule, "if ($X^TQX > c$), switch to high assurance controller", is simple and easy to verify.

Once we can ensure that the system will be kept in admissible states, statistical performance evaluations of the high performance control subsystem can be conducted safely in the actual plant operations. The plant is also protected from latent faults that tests and evaluations fail to catch.

3.4 Application Notes

The development of the high assurance controller and its recovery region satisfies the basic requirement of forward recovery – the impact caused by incorrect actions must be tolerable and recoverable. In safety critical applications, the boundary of the recovery region must be rigorously specified. On the other hand, in non-safety critical applications such as the semi-conductor wafer making, the recovery region can be codified experimentally. In this case, like the use of Hamming code in computer memory, the purpose is to reduce rather than eliminate failures. Finally, it is important to pick a sampling rate that is sufficiently fast so that the maximal error generated by incorrect control action is small during each sampling period.

Another common question is that will the simple controller unreasonably restrict the state space that can be used by the high performance controller? This turns out to be a non-problem in most applications. In the design of the controllers, there is a tradeoff between agility (control performance) and stability. Since the high performance controller often focuses on agility, its stability envelope is naturally smaller than the stability envelope of the safety controller that sacrifices performance for stability.

With the high assurance control subsystem in place, we can take advantage of the less than perfect high performance control subsystems using COTS components. Software with reasonably good quality fails only occasionally when encountering certain unusual conditions. When the software restarts under a different condition, it will work again. When the high performance control subsystem fails under unusual conditions, the high assurance control subsystem steps in until the condition becomes normal again. We can then resume the use of the high performance control subsystem.

Figure 9 illustrates the selection of control from the two control subsystems. The vertical axis represents control performance levels while the horizontal axis represents time. At time t_0, the system starts with using the high assurance control subsystem. At time t_1, under operator command, the system is switched to the new high performance control subsystem. Unfortunately, an error in the high performance control subsystem is triggered and thus the system is switched to the high assurance control subsystem at time t_2. As the high assurance control subsystem stabilizes the system, the system control is switched back to the restarted high performance control subsystem at time t_3.

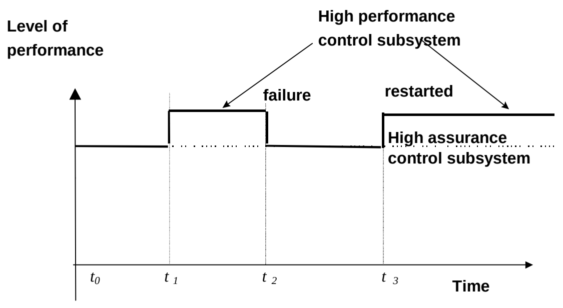
Level of performance
High performance control subsystem
failure
restarted
High assurance control subsystem
t_0
t_1
t_2
t_3
Time

Figure 9: Using Imperfect High Performance Control

Due to the existence of the high assurance control subsystem, we can in fact test newly developed high performance control software safely and reliably online in applications such as process control upgrades in factories.

4.0 Summary and Conclusion

How to improve the reliability and availability of the increasingly complex software is a serious challenge as software assumes an increasingly larger role in the critical functions of our society. It is a widely held belief that diversity in software construction entails reliability. However, we have shown that what matters most in software reliability is not the degree of diversity. Rather, it is the existence of a simple and reliable component that can be used to guarantee the critical properties.

Unfortunately, users are not satisfied with simple software. They want functionalities and features provided by complex software. Fortunately, this dilemma can be addressed by "using simplicity to control complexity". We showed this approach is applicable to linearizable control systems, which account for the vast majority of control applications.

Forward recovery using feedback is not limited to automatic control applications. For example, Ethernet rests on the idea that it is easier to correct occasional problems (packet collision) as they occur than to completely prevent them. So is the congestion control mechanism used by TCP. It is easier to correct occasional congestion than to completely avoid it. Forward recovery is also the primary tool to achieve robustness in human organizations. The endurance of democracy does not rely upon infallible leaders. Rather, it provides a mechanism to get rid of undesirable ones. Given the successful use of forward recovery with feedback in so many engineering disciplines and in human organizations, we believe that forward recovery with feedback can be applied to other types of software applications. And the notion of using simplicity to control complexity allows us to carryout the feedback process with a high degree of reliability that is associated with verifiable simple software.

Acknowledgment

This work was sponsored in part by the Office of Naval Research, in part by the Lockheed Martin Corporation and in part by Electric Power Research Institute. Many have contributed to this work. In particular, I want to thank Danbing Seto for his contributions to control theoretic development, Bruce Krogh for the semi-conductor wafer making experiment, and Michael Gagliardi for the development of the experimental demonstration systems at the SEI. I also want to thank Kris Wehner, Janek Schwarz, Xue Liu, Joao Lopes and Xiaoyan He for their contributions to the Telelab demonstration system at UIUC. Finally, I want to thank Alexander Romanovsky for discussions on the related topic of exception handling.

References

[1] Clarke, E. M., and Wing, J. M., "Formal Methods State of the Art and Future Directions", *ACM Computing Survey*, 28, 4. Dec. 1996, Pages 626 – 643.
[2] Avizienis, A., "The Methodology of N-version Programming", in *Software Fault Tolerance*, John Wiley & Sons, 1995. Editor Lyu, M. R.
[3] Randel, B., and Xu, J., "The Evolution of the Recovery Block Concept", in *Software Fault Tolerance*, John Wiley & Sons, 1995. Editor Lyu, M. R.
[4] Brilliant, S., Knight, J. C., and Leveson, N. G., "Analysis of Faults in an N-version Programming Software Experiment", *IEEE Transaction on Software Engineering*, Feb. 1990.
[5] Leveson, N. G., "Safeware: System Safety and Computers", Addison Wesley, Sept. 1994.
[6] Leveson, N. G., "Software Fault Tolerance: The Case for Forward Recovery", the proceedings of the *AIAA Conference on Computers in Aerospace*, AIAA, Hartford, Connecticut, 1983.
[7] Lyu, M. R., (Editor) "Software Fault Tolerance (Trends in Software, No. 3)", John Wiley & Sons, 1995.
[8] Yeh, Y. C. (Bob), "Dependability of the 777 Primary Flight Control System", the Proceedings of *DCCA Conference*, 1995.
[9] Sha, L., "Dependable System Upgrade", the Proceeding of *IEEE Real Time Systems Symposium*, 1998.
[10] Seto, D., Krogh, B. H., Sha, L., and Chutinan, A., "Dynamic Control System Upgrade Using Simplex Architecture", *IEEE Control*, Aug. 1998.
[11] Boyd, S., Ghaoul, L. E., Feron, E., and Balakrishnan, V., "Linear Matrix Inequality in Systems and Control Theory", *SIAM Studies in Applied Mathematics*. 1994.
[12] Seto, D., and Sha, L., "Engineering Method for Safety Region Development", CMU/SEI-99-TR-018, 1999.

Biography

Lui Sha obtained his Ph.D. from Carnegie Mellon University in 1985. He was a senior member of technical staff at Software Engineering Institute, CMU from 1986 to 1998. Since fall 1998, he is Professor of Computer Science at UIUC. He is a fellow of the IEEE, awarded "for technical leadership and research contributions that transformed real-time computing practice from an ad hoc process to an engineering process based on analytic methods." He is an associate editor of IEEE Transactions on Parallel and Distributed Systems and of the Real-Time System Journal. He served as the chair of the IEEE Real-Time Computing Technical Committee from 1999 to 2000.