Chapter 4
Separating Safety and Control Systems to Reduce Complexity

Alan Wassyng, Mark Lawford, and Tom Maibaum

4.1 Introduction

This book is about complexity in the context of analyzing, designing and implementing software intensive systems. Actually, there are three different kinds of complexity that are of direct relevance. It is thus important to define the terminology we will use so that we may be as clear as possible as to exactly what kind of complexity is under discussion at any one time.

Problem complexity—the inherent complexity of the simplest but still complete and accurate version of the application (problem) to be built.
Programming complexity—the complexity of the implementation of the application.
Computational complexity—the performance cost of an algorithm.
Complexity—if we use the generic term, 'complexity', we mean both problem and programming complexity.

At the moment there is a vast difference in what we know about the three kinds of complexity. There is a growing body of knowledge related to computational complexity, including terminology that describes how complex an algorithm is. There are also accepted measures of this kind of complexity. Unfortunately, we cannot claim the same for problem complexity and programming complexity. We speak about these (related) complexities often. We proclaim that they are an important cause of software errors. However, we do not even know how to measure them effectively, which seriously impacts our ability to design experiments to study them. Even more unfortunate is that, in the context of developing safe and dependable

T. Maibaum (✉)
McMaster University, Hamilton, ON, Canada
e-mail: tom@maibaum.org

M. Hinchey, L. Coyle (eds.), *Conquering Complexity*,
DOI 10.1007/978-1-4471-2297-5_4,

systems, it is problem complexity and programming complexity that are of primary importance.

Complexity is important to Software Engineers because we have anecdotal evidence that systems of high problem complexity are extremely difficult to build so that they are suitably dependable [15]. And we have enormous amounts of evidence that systems with high programming complexity are extremely hard to maintain, in the full general sense of maintenance. Computer Scientists and Software Engineers have spent years developing techniques for dealing with complexity. The most important of these techniques are *abstraction* and *modularization* (as a specific and somewhat limited form of separation of concerns).

Abstraction is a common and useful practice which is used to focus attention on a simplified view of the system/component. The idea is that the view should retain relevant information but ignore 'irrelevant' details that make the system/component more complex. Abstraction is an essential tool in our toolkit. It helps us understand, model and analyze complex systems. Problem complexity cannot be reduced by abstraction, though it, and some related notions, such as views, may help us cope with complex systems. What is definitely reduced by abstraction is programming complexity. Abstraction is not unique to the software world. It has been used effectively for ages by anyone who has had to build mathematical models of complex systems—physicists, engineers, economists, ecologists, and many others. Sometimes, we are so expert in abstraction that we do not notice that we have abstracted away essential details of the real system! So, abstraction can genuinely reduce complexity, but the reduction is usually temporary. At some stage, most of the details have to be reintroduced into the solution. However, we should not underestimate the usefulness of abstraction while we develop our understanding of the system that has to be built.

Modularization is a special case of *separation of concerns*. We do this, i.e., modularize, at many stages in software development. For example, we may modularize the requirements so that the required behavior is easier to understand. Typically this is done along functional lines. We can modularize the software design (and the code) so that it has some desirable properties. For example, *information hiding* was postulated by Parnas [21, 22] so that the software design would be easy to maintain under classes of foreseen changes. And, speaking of 'classes', *object oriented design/programming* was developed to further enhance our ability to modify existing design modularization when subjected to change. In all these cases, modularization has come to mean *encapsulation* of behavior and/or data in *modules*. Each module is relatively simple and the modules communicate with each other through public interfaces. This is not only an example of separation of concerns, it is also an example of an old standby in dealing with complexity—*divide-and-conquer*. Modularization lies at the very heart of modern Software Engineering. It has proved to be extremely effective in providing a mechanism for structuring software designs in particular.

Modularization has become so useful, in fact, that software experts proclaim that it is possible to ***reduce*** complexity through the use of modularization and other similar software engineering techniques and principles. We now think that this view

is flawed. There is a very good reason why it is useful to differentiate between problem complexity and programming complexity. If we are correct in supposing that there is such a concept as problem complexity, it suggests a principle we can formulate as *conservation of complexity*. Simply put, our conjecture is that we cannot reduce the programming complexity of a system to the extent that it is 'less than' the problem complexity of that system, whatever measure we use for complexity. In the case of modularization, for example, we might say that the individual program components are simplified while their interactions are made more complex. In fact, it is often observed that the (programming) complexity of modern systems is not in their components, but in the interactions between components.

So, if we cannot really reduce the programming complexity of a safety-critical system below its problem complexity, and if the dependability of the system is adversely affected by high problem/programming complexity, how can we build highly dependable safety-critical systems?

There are a number of good answers to this question—and this book contains many of them. Our answer focuses on an idea that supersedes the concept of modularization, namely separation of concerns. This approach has provided excellent solutions in a number of instances in the past. Our suggested approach is an extreme case of separation of concerns. What if we can partition the system so that we have components with no (or very little) interaction between them? For example, Canadian regulations for nuclear power generation state that safety systems in nuclear power plants have to be completely separated from the control systems in that plant, and isolated as much as possible from each other (where there is more than one safety system). Similar regulation is actually common in other countries [18, 19], as well as in the process control domain. A significant difference seems to be how strictly the regulation is enforced across countries and between the domains. A decade or so ago, there was general adherence to this principle of separation. There is now pressure to relax/remove this restriction. The pressure comes from manufacturers of these systems, *not* from regulators!

Analogous principles are used in other settings: operating systems kernels, communication kernels, etc. In recent years we have found that there are advantages in building dynamically adaptive embedded systems. These systems often have to react to malfunctions and/or changes in the environment. It seems to us that this principle of separation may be just as important for these systems as it is for many current safety-critical systems. Many adaptive and reconfigurable embedded systems integrate safety-critical and mixed-criticality components. We believe that these systems should be designed so that the safety and adaptive components must be separated for the same reasons that safety and control systems are separated. This could even cover separation of components such as those for communication from components corresponding to application features [7].

A recent paper on separation of concerns and its usefulness in relation to dependability of systems makes similar points about the usefulness of separation of concerns in relation to establishing the dependability of systems. [10]. The paper

focuses on the idea of simplicity as the underlying basis for the feasibility of establishing dependability. We revisit a few of the arguments in this paper below and add our own. Most importantly, we replace the undefinable notion of simplicity (a call to arms proclaimed for several decades by Tony Hoare [8], and now reissued by Lui Sha [24] and others), by the definable and scientific concept of ***problem*** complexity.

For the remainder of this chapter we will use separation of safety and control systems in the context of the nuclear power domain to illustrate the concepts and principles, referring to other examples as and when necessary. We first introduced the idea of *conservation of complexity* in an invited paper [27] specific to adaptive systems, which served as the basis for this chapter.

4.2 Reducing Complexity

A fundamental reason for separating control and safety systems is that we believe that, at least in the nuclear domain, fully isolated safety systems are inherently less complex than are the systems that control the reactor ("fully" here means one extreme of separation, what we might call *physical separation*). The safety subsystem is literally isolated from the control system and each safety subsystem (there were two at Darlington) is totally separated from the other. The disparity in complexity is even greater between safety systems and integrated safety and control systems. We also believe that this reduced problem complexity enables us to design, build, and certify the behavior of the safety system to a level of quality that would be difficult to achieve for an integrated, and thus more complex, system.

The safety systems at Darlington were of the order of tens of thousands of lines of code, whereas the control system was of the order of hundreds of thousands. Now, given extant criticisms of the lines of code metric for complexity, we do not want to use this essentially qualitative measure for anything other than to emphasize the difference in size and, therefore, the likely significant difference in programming complexity—and by inference, problem complexity as well. This order of magnitude difference in programming complexity alone indicates the impact on analyzability of the two pieces of software. As we know, more or less any verification approach (testing based or proof based) suffers from exponential growth in the size of the search space in relation to 'size'. Hence, the control system, and similarly an integrated control and safety system, will not be an order of magnitude more difficult to analyze, but exponentially harder.

At this point, it may be useful to discuss the principle that we have called *the conservation of complexity*. We assert that systems and their requirements have some level of inherent complexity. Sometimes, systems are designed so that they are more complex than necessary, ditto requirements. However, for a particular system, there is some level below which its complexity cannot be reduced. Principles like modularity do not reduce this inherent complexity; they simply redistribute it. Modularity may reduce complexity of parts. However, if we want to consider the complexity of the complete system we must 'add' the complexity of interactions between parts.

Modularization in the usual sense is taken to mean division into parts in relation to the functionality or features to be delivered by the application. The *divide and conquer* strategy in problem solving is often taken as the pattern on which to base such functional decompositions. What is often forgotten in such discussions is that the decomposition of a problem into subproblems that are easier to solve must be accompanied by a recomposition operation that is not 'free'. This recomposition involves some level of complexity. The complexity of interaction mentioned above is a direct reflection of this cost of recomposition. In fact, it has often been observed that the complexity of modern, large systems is down to the interaction between components, whilst components themselves tend to be trivial. Very few would argue that modern large systems are not complex, though some might argue that they have somehow reduced the complexity of the application. If there is any truth to this latter claim, it must, in our view, be related to programming complexity: surely no one would disagree with the assertion that the programming complexity of a modularized design is significantly lower than that of a monolithic design. So, this line of argument does not provide evidence for having lowered problem complexity in any way; in fact, our use of the word conservation in this context implies that it cannot be reduced.

Now, separating safety and control in a system is *not* an example of modularization in the usual sense, because, surely, we are not taming complexity by moving complexity to interaction. Separation, in this example, creates two independent systems, at least one of which is going to be inherently lower in problem complexity than that of the original problem. Of course, the other part, the control system, may also be inherently less complex than the original requirements, but the two systems, taken together, are no less complex than the original integrated system because of the conservation of complexity. This separation is an example of separation of concerns that cuts across functional hierarchies. In fact one might characterize it as doing the opposite of aspect weaving! It disentangles safety concerns from the various parts of the system and packages them up in a separate subsystem, never to be weaved again into the application.

Of course, such a complete separation may not be possible in all systems. Adaptive and dynamically reconfigurable systems may be examples of such systems. For these, we need to develop a better understanding of the separation that *is* feasible and how this contributes to a division that still enables the development of greater confidence in the safety component, because its problem complexity is significantly lower than that of the original problem and, further, its interactions with the rest of the system are also of less complexity than the original. The differences in complexity still have to be significant enough to enable the claim of simpler analysis. An example of such a system, where complete separation is not possible, is that of operating systems and trusted kernels. One of the motivations for building operating systems using trusted kernels is exactly the issue of low complexity and analyzability. The kernel is significantly simpler than the whole operating system and its interactions, usually defined through a small interface with the rest of the operating system, are also significantly less complex than interactions in the other parts of the operating system.

4.2.1 The Effect of Reduced Complexity on Quality and Dependability

In our context, it is the effect of complexity on dependability and the quality of the software that is of primary interest. Surprisingly perhaps, we have not yet in this chapter discussed any sort of definition for *'complex system'*. This is not an oversight. It seems to be a fact of life that people instinctively know what complexity means, but defining it has occupied the minds of countless philosophers and researchers from many domains over many years—and we still do not have a widely accepted definition of what constitutes a complex system. In a very recent paper, Ladyman, Lambert and Wiesner [14] list many 'definitions' of a complex system, including the following one that we found to be the most appropriate in our context. This definition originally appeared in [29]:

> "In a general sense, the adjective 'complex' describes a system or component that by design or function or both is difficult to understand and verify. [...] complexity is determined by such factors as the number of components and the intricacy of the interfaces between them, the number and intricacy of conditional branches, the degree of nesting, and the types of data structures".

This statement seems to fit our notion of programming complexity. It is directly related to the notion of "aggregate complexity", which 'concerns how individual elements work in concert to create systems with complex behavior' [16]. There have been many attempts to create practical and representative metrics for programming complexity, and some of them use the components of this definition (see [6] for representative examples). However, none has met with any significant success, and the metric most commonly used in practice is an old and simple one that we referred to earlier—*lines of code (LOC)*. There are many documented problems with using *LOC* as a metric for programming complexity [11], but alternatives seem to fare no better [5]. This brings us to our first point.

1. *Reduction in size.* The crucial fact here is that we use the resulting code size of the system as a measure of programming complexity. Size can be measured in *LOC* as discussed above. This assumes that *LOC* is typically correlated with the number of system inputs and outputs, the number of classes/modules, and even the state space of the system. Thus *LOC* provides us with an indication of programming complexity. The specific 'size' does not matter. We are interested in the size merely as an indication of the programming complexity of the system, and hence the feasibility of using rigorous (mathematical) methods and tools to complement more typical approaches, and to be able to retain sufficient intellectual control over the design and implementation of the system to achieve the required dependability. At this stage in the history of software engineering, we are capable of using formal techniques to specify the requirements and design of 'small' systems, and thus be able to mathematically verify designs against

requirements and code against designs with a level of rigor that is not yet possible for larger systems [26]. One conclusion to draw here is that reduction in programming complexity may not really be effective unless the resulting system is small enough to be amenable to a variety of validation and verification methods, not just testing. Constructing and certifying safety systems that are smaller than a hundred thousand *LOC* is a very different task compared with systems that are hundreds of thousands of *LOC*, let alone millions of *LOC*. Note that verification is just one of the activities adversely affected by the size of the system (programming complexity of the system), but it is a pivotal one.

Returning to the point at issue: if we can achieve a significant reduction in the size of the application, we believe that it is possible to reduce the *problem* complexity of that application. Put another way, the only way to reduce the size of an application by a significant amount is to reduce the problem complexity of the application. There is a trite but important assumption implicit here, and that is that the application has not been so poorly designed that we could achieve a significant reduction in programming complexity simply by doing a better job.

We believe that we can reduce the problem complexity of the system in a number of ways:

- we can scale back the number of features planned for the system;
- we may be able to reduce the number of inputs and/or outputs;
- scaling back efficiency requirements often reduces the complexity inherent in the system;
- we can require a rudimentary user interface rather than a sophisticated one;
- we can reduce or eliminate concurrency;
- we can restrict or eliminate interfaces to other systems;
- we can remove error handling;
- we can relax timing requirements.

Most readers will be quite familiar with the above list—or one very much like it. We see some or all of these actions all the time in industry. We may even have resorted to using these ‘simplifications’ ourselves. If we further examine each of these ‘cuts’, we can envision quite easily that each of them would result in a reduction in the size of the implemented system, measured by *LOC*. This would seem to confirm that these ‘cuts’ would reduce the problem complexity of the system. This fits in well with our suggestion that one way to reduce the problem complexity of a system is to partition the system. If we partition the system into two parts, for example, and if we can isolate a small, cohesive subset of the original requirements into a separate system, then that system will have significantly fewer features, inputs and/or outputs, than did the original, integrated system. There are usually two reasons for making the above ‘cuts’ to a system under development. The first is that we are far behind schedule and the schedule has to be met (not always true), so that if we do not reduce the scope of the system, we will not meet the schedule. The second is that if we try and get everything done, the quality (correctness, dependability) of the resulting system will be inadequate. In other words, experience has taught us that if we are struggling with

maintaining the quality of the system under development, reducing the number of features, inputs and/or outputs may allow us to achieve the target quality of the system. This shows that we have, for years, instinctively linked problem complexity with system dependability. The greater the complexity, the more difficult it is to achieve the required dependability.

2. *Reduction in algorithmic complexity.* Simple algorithms and data structures are easier to construct correctly in the first place, and subsequently are easier to verify as being correct. Manual verification poses few challenges and automated verification is often quite straightforward. On the other hand, proving that complex algorithms achieve desired results and that they are implemented correctly, presents us with significant challenges. This is easy to see when we examine the progress we have made in certifying scientific computation software packages. Scientific computation packages (as well as statistical packages) have a long history, going back to the 1960s. These early versions were surprisingly reliable in spite of the lack of sophistication regarding their development—by today's standards. An advantage that they enjoyed was that each method was based on strong mathematical knowledge about the algorithms and also about tests that should be performed to confirm that the methods were working correctly. As scientific computation grew more ambitious, the problem complexity of the packages grew tremendously. Today, many researchers are deeply concerned about the dependability of scientific computation [12]. The increase in algorithm complexity has led directly to an increase in problem complexity so that development and verification of large scientific computation software suites remains an open and extremely challenging research field [4]. To reduce problem complexity in a system with considerable algorithmic complexity, it is not sufficient to simply partition the system into two parts. We have to partition the system in such a way that one part will have significantly reduced algorithmic complexity. Fortunately this is possible in many of the systems we are interested in. Later, in Sect. 4.2.3, we will show why we believe that separation of safety and control is likely to result in a safety system that has much less algorithmic complexity than either the associated control system, or the integrated system.

4.2.2 Modularization and Abstraction Cannot Reduce Problem Complexity

Modularization is often touted as a way of reducing complexity. In fact modularization (and abstraction) cannot reduce problem complexity, but may actually increase programming complexity, in order to, for example, improve maintainability. Still, "conquering complexity" is a common phrase used to describe how modularization supposedly makes things simple enough for designers to be able to cope with the potential complexity of an application. The motivation for this comes from the *divide and conquer* problem solving techniques used in many areas of mathematics, engineering and science [23]. As noted above, the divide and conquer tactic is intended

to reduce the solution of some problem to the solution of several subproblems, each of which is a 'simpler' problem than the original. But an often unstated part of this tactic is the necessity to find a way of composing the solutions of the subproblems to provide the solution to the whole problem. So the overall problem complexity of the solution to the problem is a function of the complexity of the solutions to the subproblems and the complexity of the composition mechanism used to 'aggregate' the overall solution. The same may be said about programming complexity, though the function used to compute this overall complexity will likely be different from the one used for problem complexity. This function may differ from problem to problem and from one composition function to another. In modern large systems, the 'composition' operator on subproblem solutions may be extremely complex, and inherently so.

In fact, many modern systems may have little programming complexity in any particular module, but the numbers of modules and the variety of interactions and behaviors possible as a result of their combination boggle the mind. There is no obvious reduction in overall complexity as compared with the system's problem complexity. In fact, the real tactic behind the divide and conquer method is to reduce the solution of an 'unknown' to that of a number of known problems and a known technique for combining their solutions. The overt purpose of the tactic is not reduction of overall problem complexity, but a reduction in the complexity of the solution process undertaken to solve the problem—reducing the solution problem to known patterns of solutions. If (inherent) problem complexity is to mean anything, then no tactic will have the effect of reducing it. In fact, one might say that engineering methods address the issue of solution complexity—the problem of finding a solution to an application problem—by systematizing the tactics used to solve a specific class of application problems. One might conjecture that programming complexity, as discussed above, somehow reflects this solution complexity. However, we do not plan to go further in this direction in this chapter.

In respect of programming complexity, it may be conjectured that modularization techniques sometimes act to increase it. The pattern of solutions to sub problems and their composition may well act to introduce 'artificial complexities' (non-essential complexities) in relation to basic problem complexity. This is perhaps best exemplified by the problems of entanglement in object oriented implementations. As an example, in a recent investigation of a three tiered application (database, generic application software, and company specific application software), three functions of interest at the database level were potentially called by more than 80,000 functions at the generic application level, but this was again reduced to five functions at the company specific level. The enormous numbers associated with the middle layer were largely the result of the use, perhaps inappropriate, of inheritance structures. This kind of programming complexity does not appear to be uncommon in the object oriented world. We should note here that the problem of analysis in relation to dependability is clearly more a function of programming complexity than problem complexity, assuming that the former is always greater than the latter. However, problem complexity defines a minimum analysis complexity to be expected for the application.

We now come to the consideration of abstraction in relation to complexity. While modularization is often said to reduce complexity by reducing a complex system to its parts, abstraction is said to reduce complexity by 'forgetting' unnecessary details. Certainly, we would agree with this statement if the complexity referred to in the last sentence was programming complexity. The 'unnecessary details' referred to above are always intended to be those necessary to make the problem solution executable on a computer. However, it is not clear to us why abstraction should reduce problem complexity. An abstract model that captures the essence of a problem must also inherit its complexity.

Having said that, there may be one abstraction technique (and perhaps others) that appears to reduce problem complexity, namely the use of *views* or *viewpoints* [17, 20]. A view of an application is a partial specification that not only leaves out unnecessary details, but also leaves out aspects of the application problem. The view might be seen as presenting a subproblem, and the inherent problem complexity of this subproblem may well be less than that of the whole. The analysis of the view may then indeed be simpler than that of the whole. However, as for modularization, we may well have difficulties in putting views together and performing the analysis related to this 'view composition'. So we find that again, the technique does not really reduce problem complexity. The use of views is an example of separation of concerns in the more general sense discussed above. As such, when it comes to establishing dependability properties of an application, it may be quite efficacious in reducing the complexity of performing an analysis by dividing the analysis into parts that may require differing levels of rigor. An example of this will be discussed next: separating safety subsystems from control subsystems. However, for this to happen, there also has to be a commensurate reduction in programming complexity related to the core dependability concerns. If, as is usual in implementing applications, the views developed at the abstract level have no direct correspondences with parts of the application, then the programming complexity introduced by the implementation completely overwhelms the reduced complexity of individual views.

It is possible that a catastrophic example of this kind of complexity leading to disaster was the integration of patient billing information with the control of clinical X-ray therapy machines such as those reported in the articles in the New York Times [1, 2]. We have no written documentation confirming this, but have been told that this happened. Whether it is accurate or not, the possibility is very real. The medical device in question had no separate safety system; it was integrated with the control features. A very serious error occurred when the settings for the shields used to focus and aim the X-rays were accidentally left fully open leading to a serious overdose of radiation applied to a patient. Although the machine was regularly checked and calibrated, because the machine's software was directly linked to the billing system, the next time the patient came in for therapy, the device's software recovered patient information from the billing system and set the device to the configuration used in the previous overdose. So, it is possible to conjecture that a serious error imparting profound harm to the patient, which could have been prevented by a separate safety system, was compounded as a result of increased problem complexity caused by linking the device to billing subsystems. The initial error could be said to have been

caused by combining safety and control features into a complex whole, resulting in a highly complex system that was too complex for proper safety analysis. The second (and subsequent errors) were the result of making the dependability problem even more complex by introducing the link to the billing system.

4.2.3 Why Control Is More Complex than Safety

The shutdown system in a Canadian nuclear power plant is designed to monitor whether safety limits are exceeded, and in such cases to initiate the shutdown of the plant. The shutdown must be irrevocable once started, which simplifies the logic—but this principle is sometimes relaxed if the additional logic required is minimal. A nuclear reactor operates by initiating and then controlling a nuclear chain reaction. This reaction is constantly changing and so the nuclear control system algorithms initiate actions that are definitely not irrevocable. These control system algorithms are designed to keep the reactor operating within safe limits, but their purpose is to maximize productivity by maximizing the power level, and so they are far more complex than the simple checks against safety limits implemented in the shutdown systems.

The difference between control and safety systems is reflected in the mathematical analyses that are performed for these two classes of systems. The nuclear safety analysis always assumes that trips are taken to completion, and this simplifies the required behavior. The same assumption is clearly not appropriate for the control systems. Partly as a result of this assumption, in our experience, almost all the algorithms required in nuclear shutdown systems are extremely simple. This is certainly not true of the control systems. Note that we are not saying that the mathematical nuclear safety analyses performed to obtain requirements for the shutdown systems are simple. They are not, and correctness of the scientific computation code used to perform these analyses is an ongoing research topic.

There are at least two primary reductions in complexity that we expect to see in safety systems. The first is a reduction in size, and the second is a reduction in algorithmic complexity.

1. *Reduction in size.* The shutdown system is responsible for monitoring reactor attributes (neutronics, pressure, temperature, flow of coolant, etc), checking them against pre-determined limits, and initiating a shutdown if necessary. It has to be able to accept a very limited set of operator inputs, and may have limited communication functions to perform. If we use the number of lines of source code as an indication of complexity, we expect that it should be of the order of tens of thousands, and the number of system inputs and outputs under a hundred for each. These are then relatively small programs by modern standards, and tend to be more amenable to the application of rigorous software engineering techniques in ways and at a level that would not be possible for more complex systems, which typically require hundreds of thousands of *LOC*. As an example, the shutdown systems for the Darlington Nuclear Generating Station in Ontario

are of the order of 30,000 to 40,000 *LOC*. The control system for the same plant is upwards of 500,000 *LOC*. Alternatively, there may be other measures of size that are more meaningful in this context and do not correspond directly to *LOC*, but relate to complexity of analysis.

2. *Reduction in algorithmic complexity.* The control systems in nuclear power plants contain algorithms that are designed to control the nuclear chain reaction such that the plant operates at maximum power and still maintains all its monitored parameters within safe operating limits. These algorithms are also designed so that the controlled behavior is stable. By comparison, most of the algorithms in the shutdown systems are incredibly simple. A huge proportion of the algorithms implement simple checks of monitored values against predefined limits. Some of the algorithms have to cope with simple timing behaviors, while others implement very basic hysteresis behavior, and signal calibrations. The complexity of these algorithms is demonstrably orders of magnitude less than those required for the control systems.

As noted above, by reducing both size and algorithmic complexity, we have directly addressed the two main complicating factors in the analysis of software. By reducing the size of the program and by reducing algorithmic complexity, we will have reduced analysis complexity exponentially. In the ongoing battle to build dependable systems, this should be considered a signal achievement.

4.3 Separation of Concerns

There is a long-standing principle in software engineering that we can use *separation of concerns* to control complexity in software systems. Separation of control and safety systems can be viewed as a special case of separation of concerns, and there is at least one recent example in the software literature indicating that people are recognizing the importance of this [10]. Again, there is a case to be made that this separation of concerns is not the same as modularization. It is more like the splitting of the system into parts in a way that does not respect the rules of modularization. The ideas behind aspects come to mind. It seems to us that work in adaptive and reconfigurable systems has failed to consider adequately the use of such separation mechanisms to affect better control of safety functions. There is a real opportunity, in exploring these ideas, to improve safety mechanisms for this emerging class of systems.

4.3.1 Physical Separation: Reducing Complexity

A fundamental safety principle is to maintain physical separation and independence between safety systems and control systems. This helps limit the impact of common cause failures and systemic errors, and provides protection against sabotage

and cyber-attacks. These are important principles that establish the requirements to assure that high reliability requirements are met. Physical separation as a primary safety principle has been a standard requirement throughout the process control industry for decades, and *independent protection layers* are mandated in international standards such as IEC 61508 [9]. As noted above, this is also a requirement in the regulation of nuclear power plants in both Canada and the USA. The only engineering arguments against this principle come from considerations of efficiency rather than safety. However, where such an argument arises, safety always trumps efficiency. If a safe system is not efficient enough, design engineers need to find a different solution. The question of where to draw the line between integration and strict separation of safety and control systems has gained some traction in recent years. Some manufacturers of nuclear power station control systems do not wish to separate safety systems from control systems, and, compounding the problem, wish to integrate plant management systems and even billing systems into the critical software controlling the power generation. Others wish to weaken the physical and logical separation of redundant control systems by allowing communication and interaction between them, to save cost by reducing the number of parts. As a consequence, there is, unfortunately (in our opinion), a recent and deleterious trend to weakening the physical separation between shutdown systems, and between shutdown and control systems. We address this development in Sect. 4.5.

So how does this relate to our discussion on complexity? If we look again at our opening sentence in Sect. 4.2, we see that we described the separated systems as 'fully' isolated, meaning physically separated. There was a good reason for this. Physical separation of the systems helps us show that there is minimum, hopefully zero, interaction along interfaces between the systems. We need to show that any interaction between the systems is restricted to those interactions possible in their environments. This is not the same as having to cope with interactions through a common interface. To achieve this, the systems must be logically separate from each other. Demonstrating this conclusively is sometimes nontrivial. Actual physical separation makes this a much easier task. Logical connections are only possible where there are physical connections, and these would then be clearly visible—or, even better, non-existent.

As an aside, and not connected to our discussion on complexity, there are additional reasons that physical and logical separation of safety systems from each other and from control systems benefits the cause of dependability and safety.

The first of these is related to *common cause failures* [19]. Common cause failures occur when more than one component in a system fails due to a single shared cause. This is clearly not limited to software and has been studied over a significant period of time. Prevention of common cause failure is a staple of international standards and regulations related to high-dependability systems, for example, the *Common-Cause Failure Database and Analysis System: Event Data Collection, Classification, and Coding* [18], and *Guidelines on Modeling Common-Cause Failures in Probabilistic Risk Assessment* [19], nuclear regulatory documents published by the Nuclear Regulatory Commission in the USA. The *Common Cause Failure*

Database[1] is a data collection and analysis system that is used to identify, code and classify common cause failures events.

Separation on its own is not enough to prevent common cause design errors. In this case we need to add *diversity* and *independence* to our toolset. Diversity and independence are sound arguments (for software, enforced diversity [3] should be preferred), and are reflected in all international standards that apply to high-dependability systems. Diversity and independence do not make sense unless the systems are physically and conceptually separated from each other. Any commonality between the systems would serve to reduce the efficacy of these principles.

The second reason why standards and regulations mandate separation of control and safety systems is that future maintenance of an integrated system would be much more difficult. This is actually somewhat affected by the complexity of the system. Changes to the system would have to be 'guaranteed' not to adversely affect existing safety functions. If the separation between control and safety is effected through the software design/logic and not through physical and logical separation, it is much more difficult to demonstrate/prove that changes to the control system cannot affect the safety functions. A carefully constructed *information hiding design* can alleviate but cannot eliminate this concern. The situation can be made even more difficult if the control and safety systems are treated as an integrated system. These issues are particularly pertinent to adaptive and reconfigurable systems, in which the principles of separation are not well understood.

4.3.2 Ideas for Separate Safety Systems in Other Domains

We have seen that separation of control and safety is not confined to the nuclear domain. It is enforced throughout the process control industry as well. It seems clear to us that we should be considering using this principle in domains such as automotive and medical devices. Microkernels are a good example of a less drastic separation of safety and other functions. The nucleus keeps the system safe (memory checks and messaging as core functionalities) and the rest of the operating system provides the main functionality. Here we do not have physical separation, but design separation enforced through the mechanisms associated with layered architectures. Microkernels have been certified and/or verified: QNX certified for SIL3, and seL4 has been verified [13].

We have recently had occasion to consider software-driven radiation machines. These devices are effective life-savers in the fight against cancer, but they also can be devastatingly harmful if they malfunction. Two thoughts come to mind with these devices:

1. Manufacturers/vendors seem to be more concerned with including features that will help sell the devices rather than with controlling the complexity of the device so that they can be more confident that the device is fail-safe; and

[1]The US Nuclear Regulatory Commission's Common-Cause Failure Data Base (CCFDB): http://nrcoe.inel.gov/results/index.cfm?fuseaction=CCFDB.showMenu.

2. It should be possible to add a low-complexity safety system that will 'guarantee' that the device does not deliver an overdose to any patient.

The safety system could, for example, require simple inputs from the doctor that limit the allowable dosage for a specific patient, and then monitor the radiation to ensure this dosage is not exceeded. This safety system would be completely independent of the control system that 'drives' the device. It would also be independent of any billing system that might compromise safety features, preventing accidents such as the ones noted above.

There are currently a number of *active safety* functions included in modern cars. These include automatic braking, adaptive cruise control, lane departure warning systems, adaptive high beam and adaptive headlamps. Typically, these are implemented as self-contained, isolated units, although some of them clearly have to be integrated with other functions—braking for instance. Although the auto industry seems to have realized that keeping such components as isolated as possible helps to deal with complexity issues and increases our ability to engineer extremely dependable systems, this objective is undermined by the need to interconnect some subsystems, e.g., braking and throttle subsystems, and the fact that subsystems may share processors and communication buses with other subsystems. It may be that we can further improve the dependability and maintainability of the systems by isolating safety from control again, rather than by relying on functional modularization.

4.4 Reducing Programming Complexity: The Engineering Approach

Engineers are continually faced with the issue of problem complexity and its impact on engineering design. For most situations met by engineers in their every day work, engineers have developed a way of dealing with this issue: the engineering method, or what Vincenti calls *normal design* [25]. Over time, as engineers solve specific problems in some domain, the successful approaches are incorporated into a standard engineering method specific for those kinds of *devices* [25]. Devices in this sense are the subject of normal design methods. Engineers know that if they follow the prescriptions of the method, including which analyses to do when and which decision to make in light of results of analysis, they are likely to design a safe and effective product. As we have noted elsewhere [28], this also forms the basis of the prescriptive regulatory regimes in classical engineering. Radical design involves design problems that are not within the normal envelope associated with a normal design method. Some new element is introduced, e.g., untried technology, or some new combination of technologies, which takes the design problem outside the incremental improvement normal design supports. This makes the achievement of safe and effective designs more problematic and requires much more serious attention to justification of safety properties. From the point of view of problem complexity, normal design helps to tame this complexity, but not reduce it, by systematizing

standard solutions to design problems. In analogy with divide and conquer techniques, the motivation behind normal design is not that of reducing problem complexity, but the reduction of programming complexity. This also sheds some light on the ongoing discussion of process based standards in software certification versus product based standards [28]. Engineers put a lot of store in normal design methods providing a higher level of assurance of safety and effectiveness of products. A process based standard for software development standardizes the process to be used in developing a new software product, but does not propose a normal design method for software, either generally or for a specific domain. This is the missing ingredient required to enable a process based claim for the product to be safe and/or effective. Until such process standards evolve to be the equivalent of normal design methods, we cannot give them much credit for reducing programming complexity, and such process based claims probably should be mistrusted.

One of the principles we would expect/hope to see in a software process standard based on normal design, is the guidance for how to separate control and safety systems so as to reduce the problem complexity of the safety system.

4.5 Conclusion

Separation of control and safety systems can be viewed as a special case of separation of concerns. This is not the same as modularization. It is a strict partitioning of the system into at least two parts, one of which contains the safety related behavior. The idea is that the separated and isolated safety system will have lower problem complexity than would the integrated system. Unlike the dangerous practice in aspect oriented programming, it is not our intention to weave the separated concern back into the application software.

We believe that separation of control systems and safety systems in the nuclear power industry is not only a good principle to follow, but that rigorous adherence to this principle should make it possible to analyze the system to an extent where we develop much greater confidence in the safety of the plant. The reasons are presented above, but the primary reason is that the reduction in complexity allows us to employ techniques that currently would not be possible for more complex systems. Without these mathematically based techniques we would be reduced to relying on testing alone to show conformance with requirements and correctness. It would also be much more difficult to apply techniques such as model checking, to confirm safe behavior at the requirements level. Recent trends in the nuclear industry would seem to indicate that manufacturers wish to abandon, at least to some degree, the need for separation of safety and control functions, and, arguably even worse, they want to abandon the basic principle of physical and logical separation between replicated safety functions. This trend is dangerous, because it moves complexity from elsewhere in the system, back into the safety function, thus significantly increasing the complexity of the safety function without significant reduction in the complexity of the control function. There appears to be no gain here, except an economic one. We are concerned that manufacturers seem to think that one time cost savings in the

original development of these systems would be more important than the increased assurance we could realize in the dependability and safety of these systems. In fact, it is quite likely that adherence to this principle of separation will result in a long-term cost reduction, since the safety components in the overall system will be less likely to require corrective modification over the life of the system. Other modifications/enhancements can typically be made with reduced re-verification since the simpler safety systems can be pre-verified with ranges for constants, and information hiding designs on these smaller systems can help us prove the localization of changes.

The nuclear power domain is but one example domain in which this technique of separating control and safety should be common practice—preferably mandated by regulatory authorities. It also seems clear to us, that this same principle can be applied to building highly dependable, *cyber-physical systems*, such as medical devices and 'smarter cars'.

Acknowledgements This work is supported by the Ontario Research Fund, and the National Science and Engineering Research Council of Canada.

References

1. Bogdanich, W.: Radiation offers new cures, and ways to do harm. The New York Times Online (2010). Published January 23, 2010. Available online: http://www.nytimes.com/2010/01/24/health/24radiation.html
2. Bogdanich, W., Rebelo, K.: A pinpoint beam strays invisibly, harming instead of healing. The New York Times Online (2010). Published December 28, 2010. Available online: http://www.nytimes.com/2010/12/29/health/29radiation.html
3. Caglayan, A., Lorczak, P., Eckhardt, D.: An experimental investigation of software diversity in a fault-tolerant avionics application. In: Proceedings Seventh Symposium on Reliable Distributed Systems, pp. 63–70 (1988)
4. Easterbrook, S., Johns, T.: Engineering the software for understanding climate change. Comput. Sci. Eng. **11**(6), 65–74 (2009)
5. Fenton, N., Neil, M.: Software metrics: successes, failures and new directions. J. Syst. Softw. **47**(2–3), 149–157 (1999)
6. Fenton, N.E., Pfleeger, S.L.: Software Metrics: A Rigorous and Practical Approach. PWS Publishing Co., Boston (1998)
7. Fischmeister, S., Sokolsky, O., Lee, I.: A verifiable language for programming real-time communication schedules. IEEE Transactions on Computers 1505–1519 (2007)
8. Hoare, C.A.R.: The emperor's old clothes. Commun. ACM **24**(2), 75–83 (1981)
9. IEC 61508: Functional safety of electrical/electronic/programmable electronic (E/E/EP) safety-related systems: Parts 3 and 7. International Electrotechnical Commission (IEC) (2010)
10. Jackson, D., Kang, E.: Separation of concerns for dependable software design. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER'10, pp. 173–176. ACM, New York (2010)
11. Jones, C.: Software metrics: good, bad and missing. Computer **27**(9), 98–100 (1994)
12. Kelly, D.F.: A software chasm: software engineering and scientific computing. IEEE Softw. **24**(6), 119–120 (2007)
13. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, pp. 207–220. ACM, New York (2009)

14. Ladyman, J., Lambert, J., Wiesner, K.: What is a complex system? http://philsci-archive.pitt.edu/8496/ (2011). Preprint
15. Lee, L.: The Day the Phones Stopped. Donald I. Fine Inc., New York (1991)
16. Manson, S.M.: Simplifying complexity: a review of complexity theory. Geoforum **32**(3), 405–414 (2001)
17. Niskier, C., Maibaum, T., Schwabe, D.: A pluralistic knowledge-based approach to software specification. In: Ghezzi, C., McDermid, J. (eds.) ESEC '89. Lecture Notes in Computer Science, vol. 387, pp. 411–423. Springer, Berlin (1989)
18. NRC Staff: Common-cause failure database and analysis system: event data collection, classification, and coding. Tech. rep. NUREG/CR-6268, US Nuclear Regulatory Commission (1998)
19. NRC Staff: Guidelines on modeling common-cause failures in probabilistic risk assessment. Tech. rep. NUREG/CR-5485, US Nuclear Regulatory Commission (1998)
20. Nuseibeh, B., Kramer, J., Finkelstein, A.: A framework for expressing the relationships between multiple views in requirements specification. IEEE Trans. Softw. Eng. **20**, 760–773 (1994)
21. Parnas, D.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)
22. Parnas, D.L., Clements, P.C., Weiss, D.M.: The modular structure of complex systems. IEEE Trans. Softw. Eng. **SE-11**(3), 66–259 (1985)
23. Polya, G., Stewart, I.: How to Solve It. Princeton University Press, Princeton (1948)
24. Sha, L.: Using simplicity to control complexity. IEEE Software, 20–28 (2001). http://doi.ieeecomputersociety.org/10.1109/MS.2001.936213
25. Vincenti, W.G.: What Engineers Know and how They Know It: Analytical Studies from Aeronautical History. Johns Hopkins University Press, Baltimore (1993)
26. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: International Symposium of Formal Methods Europe Proceedings. Lecture Notes in Computer Science, vol. 2805, pp. 133–153. Springer, Pisa (2003)
27. Wassyng, A., Lawford, M., Maibaum, T., Luxat, J.: Separation of control and safety systems. In: Fischmeister, S., Phan, L.T. (eds.) APRES'11: Adaptive and Reconfigurable Embedded Systems, Chicago, IL, pp. 11–14 (2011)
28. Wassyng, A., Maibaum, T., Lawford, M.: On software certification: we need product-focused approaches. In: Choppy, C., Sokolsky, O. (eds.) Foundations of Computer Software. Future Trends and Techniques for Development. Lecture Notes in Computer Science, vol. 6028, pp. 250–274. Springer, Berlin (2010)
29. Weng, G., Bhalla, U., Iyengar, R.: Complexity in biological signaling systems. Science **284**(5411), 92 (1999)