Safety Tactics for Software Architecture Design

Weihang Wu Tim Kelly
Department of Computer Science, University of York, York YO10 5DD, UK
{weihang.wu, tim.kelly}@cs.york.ac.uk

Abstract

The influence of architecture in assurance of system safety is being increasingly recognised in mission-critical software applications. Nevertheless, most architectural strategies have not been developed to the extent necessary to ensure safety of these systems. Moreover, many software safety standards fail to discuss the rationale behind the adoption of alternative architectural mechanisms. Safety has not been explicitly considered by existing software architecture design methodologies. As a result, there is little practical guidance on how to address safety concerns in 'shaping' a 'safe' software architecture.

This paper presents a method for software architecture design within the context of safety. This method is centred upon extending the existing notion of architectural tactics to include safety as a consideration. The approach extends existing software architecture design methodologies and demonstrates the true value of deployment of specific protection mechanisms. The feasibility of this method is demonstrated by an example.

1. Introduction

The significance of architecture-based development has been increasingly recognised in mission-critical industries. The safety-critical standard IEC 61508 Part 3 [8] states that:

> "From a safety viewpoint, the software architecture is where the basic safety strategy is developed in the software."

However, there exists little guidance on how to develop such a basic safety strategy in software architecture design. Three key issues need to be addressed in order to improve this situation:

- **Design techniques.** A vast number of software safety design techniques have emerged in both research and practice. Deciding upon appropriate techniques or mechanisms to be employed in the context of a particular system is architectural and thus is crucial to the whole software development process. IEC 61508 has provided some general guidance on the selection of these techniques according to the safety criticality of a software element under design. However, this guidance fails to demonstrate further how to exploit these suggested techniques to maximise the protection against failures.
- **Architectural patterns.** Over the last decade, the notion of *pattern* or *style* has been widely adopted to aid software architecture design. An architectural pattern defines the types of components and connectors and a set of constraints on how instances of these types can be combined in a software system [13]. Patterns also influence architecting dependable systems [10]. Example of safety-related patterns include Triple Modular Redundancy (TMR) [15] and Idealised C2 [10]. However, the coarse-grain nature of patterns can make it difficult to reason precisely about the safety properties achieved. Figure 1 depicts a TMR pattern and a possible variation. Although both patterns share much in common, they exhibit different safety properties. In order to avoid defining excessive number of variant patterns in this way, it is attractive to gain a more fundamental understanding of the design principles from which patterns and their variants can be created.

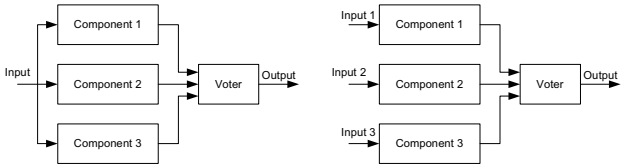


Figure 1. A TMR pattern and a variation

- **Safety analysis.** Although a considerable number of safety analysis techniques have been proposed to aid software design such as Software Hazard Analysis and Resolution in Design (SHARD) [12], there is little analysis work focusing on an architectural level to aid software architecture design. In particular,

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

safety is the entire property of a system; it is almost impossible to analyse software safety effectively without considering system or platform safety. We thus need a safety analysis approach that is able to model the integration of software with hardware or other system components and to characterise the architectural elements at different architectural levels.

This paper describes an approach to software architecture design for safety-related systems. Our work was inspired by the notion of architectural tactic [4] proposed by the Software Engineering Institute (SEI) at Carnegie Mellon University. The principle of architectural tactics is to identify and codify the underlying primitives of patterns in order to solve the problem of the intractable number of patterns existing [3]. Clearly, there are relatively fewer tactics to be handled, since there are many ways in which tactics can be combined into patterns. The SEI has proposed tactics for six quality attributes (availability, modifiability, security, performance, usability, and testability) [4]. This is inadequate, however, in order to meet the need of all the attribute communities. Safety stands out as an emergent property of a system that should be considered within the safety domain. Based upon the work of architectural tactics, it is natural to introduce the notion of safety tactics that connect software safety and software architecture.

The following sections provide a discussion of our tactical approach to address the above three issues. This tactical approach extends the existing architectural tactic development framework proposed in [1, 2], with a view to include consideration of safety as a quality attribute. The approach can be divided into four main sections. Section 2 defines an analytic model to analyse software safety at architectural level. Section 3 presents the development of safety tactics. Section 4 illustrates the tactic-based approach for software architecture design. Finally, section 5 demonstrates this approach by means of an example.

2. An analytic safety-attribute model

Safety design concerns the identification and management of hazards. Hazards are caused by failures. A distinction is often made between causes of failures in physical devices (e.g., *random failures*) and failures in software. Software does not fail randomly; its "failures" are due to its *systematic* nature (e.g., design faults). In software systems, safety is thus achieved by avoiding, or protecting against, these failures. As a result, the focus of attention in our analytic model is the relationship between the safety attribute and software architecture with respect to failures. Four key elements are thus identified to analyse this relationship: *failure classification*, *failure cause*, *failure behaviour*, and *failure property*. These four elements form an analytic model for the safety quality attribute and thus prompt our consideration of the effective measures against these projected failures. Figure 2 shows this analytic model in a simple notation, which demonstrates how software-level failures can contribute to the system-level behaviour. This model is defined in terms of software system component and other system components. The underlying computer hardware is also distinguished from the software, since unintended behaviour of computer hardware can lead to the hazardous execution of software. Finally, the failure behaviour is analysed with respect to the interaction of (software) components.

2.1 Failure classification

Systematic failures in software can arise from any stage of the development process, which makes the work of failure identification almost infeasible. Thus, a better approach is to analyse them and to classify them at an abstract level. Several classifications of failure types exist in the literature [6, 14, 12]. We have adopted Pumfrey's failure classification [12] as the most appropriate framework. The failure categories we have adopted are thus defined as follows:

- **Service provision:** *Omission* (the failure of an event to occur), *Commission* (the spurious occurrence of an event).
- **Service timing:** *Early* (the occurrence of an event before the time required), *Late* (the occurrence of an event after the time required).
- **Service value:** *Coarse incorrect* (incorrect but detectable value delivered), *Subtle incorrect* (incorrect and undetectable value delivered).

2.2 Failure cause

The second stage in analysing failures concerns how are they generated among architectural elements. Again, to make the identification work feasible, we also group all potential failure causes into three abstract types: *software itself*, *its underlying hardware*, and *its environment*, as shown in Figure 2.

- **The software itself** – Software including its underlying operating system is an abstraction. It can only contain flaws arising from either incomplete or inaccurate specification, or incorrect design and implementation. These flaws can lead to unexpected behaviour that the software exhibits within a system.
- **Underlying hardware –** Even correct software can still fail due to unexpected behaviours of the underlying hardware.
- **Environment –** By treating software as a black box,

we are also interested in anomalies that originate externally to the software and are fed into the software as inputs. Leveson [11] describes such anomalies as *environment disturbance*.

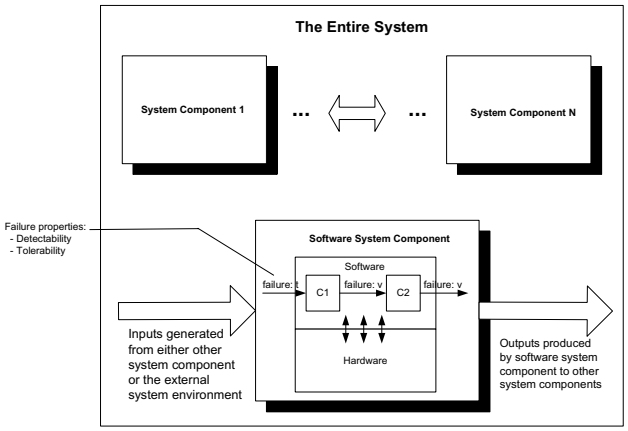


Figure 2. A simple notation for a safety model

2.3 Failure behaviour

An architectural element cannot only generate failures, but also propagate failures or transform them into another type. The analysis of failure behaviour prompts our consideration of how to select appropriate one or more architectural mechanisms to handle anticipated failures safely. There are two forms of failure behaviour in our analytic model: *failure propagation*, and *failure transformation* [7].

- **Failure propagation –** A good way to illustrate this failure behaviour is using a simple example. Consider the example in Figure 2, assuming that the component C2 is responsible for computing the data from the component C1 and outputting the result of the computation. If C1 produces an incorrect data for any reason (say, a value failure arrives at C1), C2 will also produce a wrong result given that there is no protection mechanism employed in C2 (say, a value failure now arrives at C2). In this case, it's easy to see that the value failure of C1 has propagated to C2.
- **Failure transformation –** For the same example in Figure 2, now assume that the component C1 is used to collect the data from the sensors. If the data from sensors arrives late (say, a timing failure arrives at the input of C1), C1 will simply collect the out-of-date data and produce an incorrect output given that there is no detection mechanism employed in C1 (say, a value failure arrives at the output of C1). In this instance, the timing failure has been transformed into a value failure.

2.4 Failure property

The final consideration in our analytic model concerns the properties of failures that can be used to guide the selection of effective protection mechanisms. Two such useful properties are *detectability* [6], and *tolerability* [14].

- **Detectability –** When treating an architectural element as the provider of a service or a sequence of services, failures can be regarded as detectable or undetectable by its service recipient (another architectural element). Not all failures are detectable; some failures are inherently undetectable (at least within a specific system boundary) [12]. Most protection mechanisms rely heavily upon the detection of a failure.
- **Tolerability –** Failures can be tolerated by minimising their effects through the provision of appropriate protection mechanisms.

These two properties motivate the organisation of safety tactics defined in section 3.

3. Safety tactics

This section describes the development of safety tactics, which focuses on three aspects: elicitation, organisation and documentation.

3.1 Elicitation

The elicitation of safety tactics was based upon a broad survey of existing software safety design techniques used in both research and practice (e.g., [5]). The selection was restricted by considering their suitability for architectural design. Defensive programming, for instance, is an important software safety design technique, but it is irrelevant for architectural design, since it is focused at the implementation level. One of the obstacles to the elicitation work is the complicated relationship between techniques and tactics. A technique may implement multiple tactics of a quality attribute. Also a technique may package mechanisms that are not related to quality attributes. The main tasks of elicitation heavily rely on our analysis of the underlying protection mechanisms of each design technique. We have elicited seventeen safety tactics, which is sufficient to demonstrate a starting point for architectural design in the safety domain.

3.2 Organisation

Based upon our analytic safety model, the architectural parameters that safety tactics control can be specified in

terms of failure elimination, failure detection, and failure tolerance. Safety tactics can thus be organised into three sets: tactics for *failure avoidance*, tactics for *failure detection*, and tactics for *failure containment*. To be consistent with the SEI's tactics work, we also organised safety tactics as a hierarchy, as shown in Figure 3.

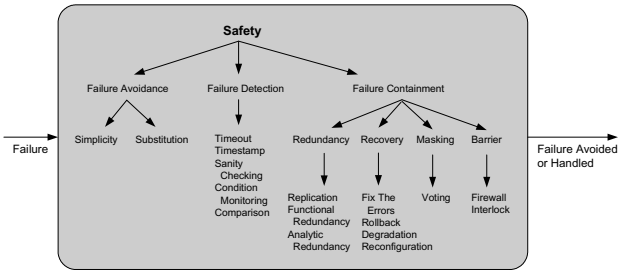


Figure 3. A hierarchy of safety tactics

3.3 Documentation

We defined a template for the documentation of safety tactics. This template offers a number of advantages over the free-text description presented by the SEI. Firstly, tactics are documented in a structured way that can facilitate comparison and analysis. Secondly, this structured approach highlights the important information (e.g., rationale) that needs to be captured in a tactic. The tactic template used for safety tactics documentation is shown in Table 1. An example documented safety tactic, *rollback*, is shown in Table 2.

Table 1. A proposed template of safety tactics

Tactic Name	
Aim	A sentence to summarise the main purpose of this tactic
Description	A description of this tactic including the means used to meet the stated aim
Rationale	Claims for this tactic that can assist in achieving the safety goal, which are expressed with the Goal Structuring Notation (GSN) [9]
Applicability	General rules and circumstances for the use of this tactic regarding any assumption, cost and other factors
Consequences	The expected system response to the arrival of failures
Side Effects	The side effects on other common quality attributes
Practical Strategies	Some examples of practical strategies derived from this tactic
Patterns	Any patterns that implement this tactic
Related Tactics	Alternative or complementary tactics

4. Deriving safety tactics

This section describes how safety tactics can be applied in software architecture and architectural patterns definition.

4.1 Properties of safety tactics

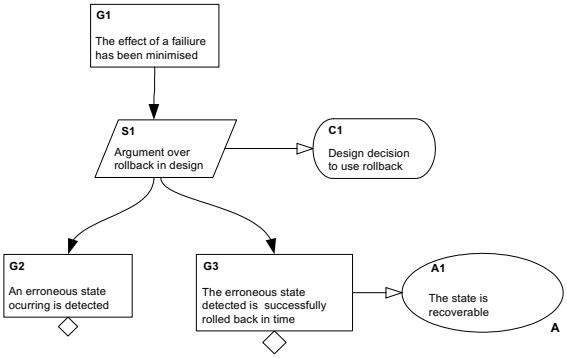
It should be noted that some of the properties are very general and can be applicable to the entire set of safety tactics, or even to other quality-attribute tactics, and others are only specific to particular safety tactics.

- **Safety tactics can be refined.** Like other quality-attribute tactics, safety tactics can exist at various levels of abstraction. For an example of *redundancy*, it can be refined into three different forms: *replication, functional redundancy,* or *analytic redundancy*.
- **Safety tactics are not necessarily independent.** The application of one tactic may assume that another tactic has already been applied. A typical example is *voting*. The application of *voting* may assume that some form of redundancy has existed in the system.
- **Safety tactics can be combined.** In general, each safety tactic can only address a specific aspect of safety. However, safety tactics can be combined to address multiple aspects of safety. Again, for the example of *voting*, it has been known that it can mask the effect of some faulty inputs. But it also has the weak point, which is the voter itself that can be vulnerable to single-point failure. To enhance the degree of safety of this tactic, it is possible to apply *redundancy* to *voting* by introducing additional voter elements, thereby removing the potential source of single-point failure.
- **Safety tactics can be implemented by either software or hardware.** In many cases, an architect has a choice between software and hardware tactic schemes. A simple example is *voting*, which can be implemented either in hardware or in software. For a hardware-implemented voter, it may only require a combination of simple single-bit circuits. For a software-implemented voter, it has its own costs in terms of development efforts, speed of operation and an increase in system complexity. The primary considerations here are always based on a trade-off among cost, performance and complexity.

4.2 Using safety tactics to construct patterns

It should be possible to construct safety-related architectural patterns in the context of a scenario, based upon the collection of safety tactics. Here we chose a

Table 2. An example documented safety tactic

<table>
<tr><th colspan="2">Rollback</th></tr>
<tr><td>Aim</td><td rowspan="2">Applicability
The tactic can be applied when the system state is recoverable. It is always combined with tactics for failure detection for use in order to activate the recovery procedure. This tactic is more secure as it places no reliance on the current system state, and it is less expensive than other failure recovery tactics such as degradation and reconfiguration and tactics for failure masking. But it may not be suitable for timing-critical systems since re-execution will probably miss the deadline requirements of these systems. Furthermore, successful rollback does not mean that faults have been eliminated; failure may still recreate when rolling back and retry again in a new time.</td></tr>
<tr><td>To roll back in time to a previous known state upon the detection of an erroneous sate and then to retry execution.</td></tr>
<tr><td>Description</td><td rowspan="2"></td></tr>
<tr><td>Also known as “recovery-and-retry”, it simply attempts to simulate the reversal of time, in the hope that the earlier known state in the new time will not recreate the failure. The tactic is applied when a failure is detected. In practice, this tactic can be applied to multi-version software with the combination of redundancy tactics. In such situations, rollback retries computation using an alternate version of software that is assumed to work better than the original one.</td></tr>
<tr><td>Rationale</td><td>Consequences</td></tr>
<tr><td>The purpose of this tactic is to contain failure. This forms the basis for presenting claims about this tactic.
 
</td><td>The erroneous state leading to a failure is repaired. But this failure may not be contained successfully and it may recreate again after rolling back.
Side Effects
It will incur penalties both in the speed of system operation and in the expenditure of resources needed to permit state restoration.
Practical Strategies
Recovery block
Patterns
Simplex pattern
Related Tactics
The same set of safety tactics for recovery are:
Fix the errors, degradation, and reconfiguration
This tactic is often combined with tactics for failure detection in use</td></tr>
</table>

simplified Control/Monitor scenario as an example: there is a basic safety requirement that the software must monitor any potentially hazardous conditions so that proper protection mechanisms can be executed, although there are not detailed requirements about what exactly should be monitored. This must be addressed in the architectural design. If we examine the available safety tactics in section 3 in light of the above analysis, we will find that a number of tactics can be used: detection, barrier, redundancy, and masking (again, these tactics can be refined into more concrete forms). Different combinations of these identified tactics will construct different forms of the control/monitor pattern. We here chose two example control/monitor patterns derived from various combinations of these tactics. Notably, the two patterns are not the only patterns that can be derived but illustrate two plausible examples.

Control/Monitor Pattern 1

The following tactics may be chosen at this point:

- *Condition Monitoring*. It is used to detect any malfunction of the actuator by monitoring either the performance of actuator or specific conditions.
- *Functional Redundancy.* We may introduce the additional independent version of the control function. This significantly limits the opportunities for failures of the control function.
- *Comparison.* It should be used to detect any discrepancy between the two independent versions of the control function by a comparison of their outputs. The tactic for *stop* is associated with it.

- *Interlock.* Correct sequencing of events should be enforced. For example, the control function must not produce the next result value without the previous 'ok' message from the monitor component.

Then we may allocate responsibilities to the architectural elements of the pattern that are associated with the selected safety tactics.

Control/Monitor Pattern 2

In the above pattern, the responsibility of failure detection completely relies on the monitor component. And no protection is taken against its failure, which leads to a potential source of single-point failure in the system. Furthermore, there is a strong dependency between the control and monitoring functions, since they both are implemented in the same "Monitor" component. Thus, we may instead allocate the safety responsibility to both the "Monitor" and "Control" channels by applying the tactic for *comparison* to each component to form cross-checking. Once these two channels do not agree with each other, action should be taken by the "Output" component to ensure that the actuator is stopped in time. Figure 4 shows the Control/Monitor Patterns 1 and 2.

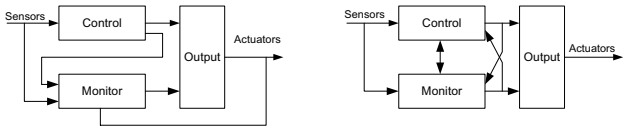


Figure 4. The Control/Monitor Patterns 1 and 2

5. An example

Our example concerns the design of the control system of an Automatic Guided Vehicle (AGV). The favoured system consists of a number of vehicles that navigate using a scanning laser and a loaded map file. The movement of each vehicle is instructed by its embedded control system that calculates its position either precisely by triangulation or by dead-reckoning or a combination of both. Figure 5 shows the collision detection mechanisms proposed for each vehicle. The braking mechanisms are hydraulically operated. There are two sensing systems:

- A proximity sensing system where proximity sensors signal to the control system that an object is being approached, with sufficient space to stop the vehicle before the bumpers touch the object
- Bumpers whose pressure is detected by microswitches, which will open if force is applied to the bumpers, cutting off the power to the drive motor regardless of any action taken by the control system.

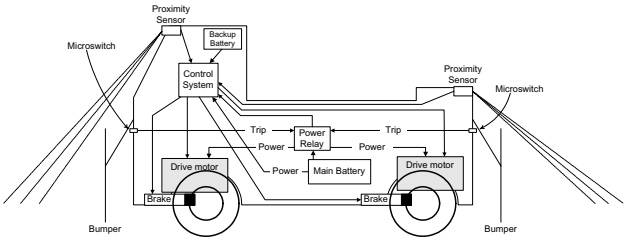


Figure 5. AGV collision detection mechanisms

5.1 Elicitation of software safety requirements

Our software safety design process always starts with the system/platform hazards identified by preliminary hazard analysis. Here we considered only one of the main hazards arising from AGV movement operation: *AGV systems faulty such that obstacle detection and/or braking does not operate correctly*. Thus one of the AGV platform safety requirements is that *AGV platform shall avoid collision*. Our design process precedes by refining this platform safety requirement into the lower-level system functional safety requirements: *AGV shall detect obstacle and stop in time*. Further refinement of this system safety function requires the allocation of stopping function to hardware or software. The *Substitution* tactic tells us using simple hardware protection mechanism is safer. However, there is a domain constraint that revising hardware design is not allowed. Therefore a software safety requirement is elicited during the design process: *control system shall output commands in time in response to obstacle detection*. The whole safety requirements and design processes are recorded by the Goal Structuring Notation (GSN) [9]. The above top-down safety requirements refinement process with hardware/software tradeoffs is illustrated in Figure 6, 7.

5.2 Identification of possible safety tactics

The elicited high-level software safety requirements can be further refined into lower-level software functional safety requirement and timing requirement. Let us consider the software functional safety requirement: *control system shall always output brake and stop commands in response to detection even in the presence of credible failures*. Since failures are unavoidable, tactics must be selected from the categories of failure detection and failure containment. Obviously, *time-out* and *timestamp* tactics are irrelevant in this case. The *sanity-checking* tactic is also rejected, since there is nothing to act on the validity signal. Moreover, the *recovery* tactics set are deemed to be inappropriate simply because any recovery action cannot immediately stop collision. Finally, the *barrier* tactics set are not suitable candidates since they are used as passive protection mechanisms. Therefore, possible safety tactic alternatives would be:

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

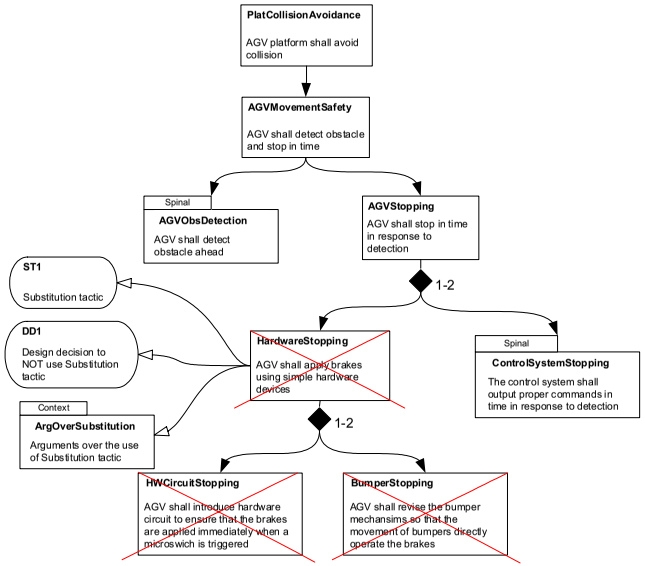


Figure 6. The top-down process for eliciting software safety requirements

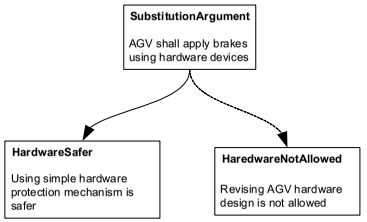


Figure 7. Arguments over the use of substitution

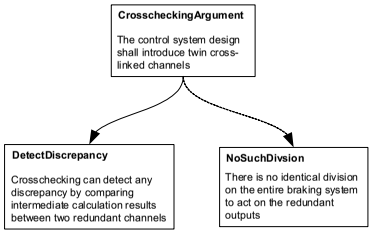


Figure 8. Arguments over the use of crosschecking

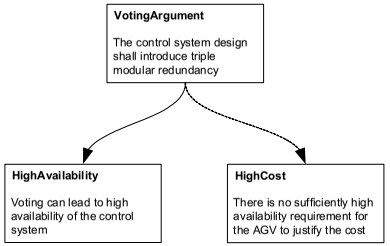


Figure 9. Arguments over the use of voting

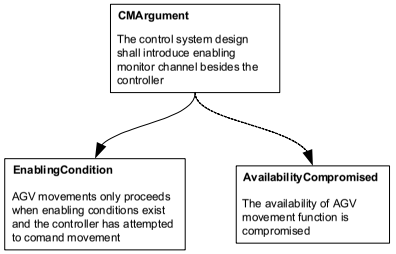


Figure 10. Arguments over the use of enabling monitor

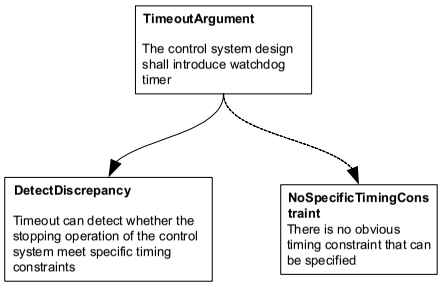


Figure 11. Arguments over the use of timeout

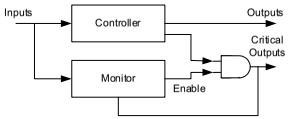


Figure 13. The initial design of control system (software) architecture

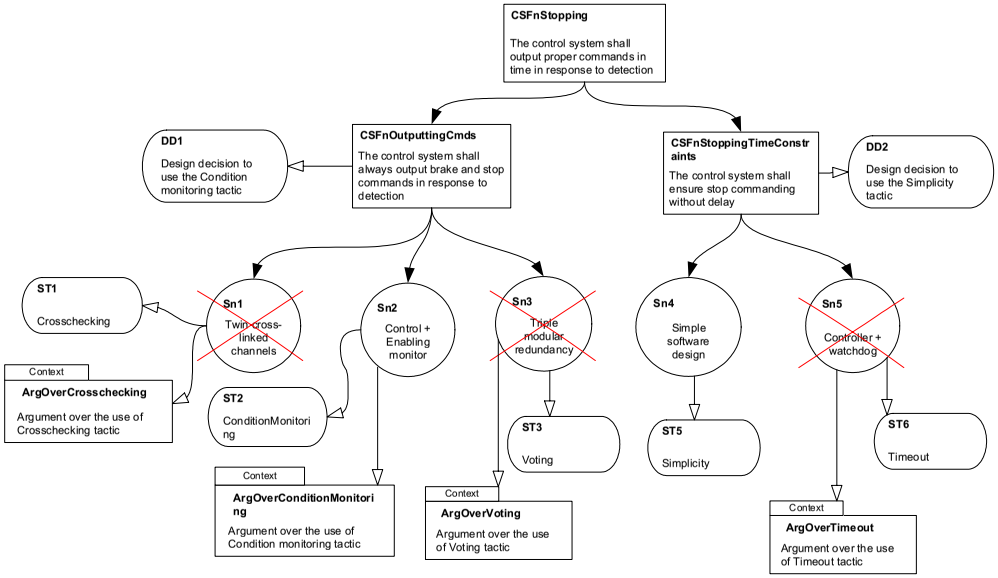


Figure 12. The top-down process for selecting safety tactics

condition monitoring, *comparison*, *voting* and *redundancy*. Clearly, *redundancy* cannot be applied alone, and actually the implementation of the previous three alternatives requires some form of redundancy. In other words, the possible tactic choices would be: *crosschecking* (*comparison* and *functional redundancy*), *enabling monito*r (with *analytic redundancy*), and *voting* (with *functional redundancy*). Likewise, in order to meet the timing requirement of stopping function, we identified two possible tactics: *simplicity* and *timeout*.

5.3 Selection and combination of safety tactics

The selection of appropriate tactics is based on the construction of arguments for each candidate. These arguments can make positive or negative contributions to our decisions, which are separately derived from design rationale, domain constraints, cost factors. The arguments for or against each possible safety tactic identified are shown in Figure 8, 9, 10 and 11. The designer is aided in their decision-making by referring to these argument structures. The process of the selection is illustrated in Figure 12.

5.4 Relating safety tactics to architectural design

Once we've chosen appropriate safety tactics, we can compose the architecture fragments. The result is shown in Figure 13.

6. Conclusions

This paper illustrates a safety-directed method for software architecture design, which differs from reliability-driven approaches in the following two aspects:

- It focuses on addressing not only the failures of components but also failure behaviours through the interaction of components.
- It treats safety requirements as an important element in the safety design process.

This method is based upon the recent SEI's work of architectural tactics to include safety as a consideration. Specifically, this extension provides the following contributions to existing software safety design practices:

- It codifies a collection of well-documented safety-related design decisions to aid designers in selecting appropriate techniques to achieve safety requirements.
- It provides practical guidance on how to select/construct safety-related software architecture patterns/fragments by deriving safety tactics.
- It defines an analytic safety model that can examine software safety alongside hardware during design decomposition.

References

[1] Bachmann, F., Bass, L. & Klein, M., "Illuminating the Fundamental Contributors to Software Architecture Quality", Technical Report, CMU/SEI-2002-TR-025, Software Engineering Institute, Carnegie Mellon University, August 2002.

[2] Bachmann, F., Bass, L. & Klein, M., "Deriving Architectural Tactics: A Step Toward Methodical Architectural Design", Technical Report, CMU/SEI-2003-TR-004, Software Engineering Institute, Carnegie Mellon University, March 2003.

[3] Bass, L., Klein, M. & Bachmann, F., "Quality Attribute Design Primitives", Technical Report, CMU/SEI-2000-TN-017, Software Engineering Institute, Carnegie Mellon University, December 2000.

[4] Bass, L., Clements, P. C. & Kazman, R., *Software Architecture in Practice*, 2nd edition, Addison-Wesley, London, April 2003.

[5] Bishop, P. G., *Dependability of Critical Computer Systems 3: Techniques Directory*, Elsevier Applied Science, London, 1990.

[6] Bondavalli, A. & Simoncini, L., "Failure Classification with respect to Detection", in *First Year's Report*, Task B: Specification and Design Dependability, ESPIRIT BRA Project 3092: Predictably Dependable Computing Systems, May 1990.

[7] Fenelon, P., "Failure Propagation and Transformation Notation – Handbook and User Guide", High Integrity Systems Engineering Group, Technical Report, Department of Computer Science, University of York, October 1993.

[8] IEC, "615038 – Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems", International Electrotechnical Commission, 1998.

[9] Kelly, T., "Arguing Safety – A Systematic Approach to Managing Safety Cases", DPhil Thesis, YCST-99-05, Department of Computer Science, University of York, York, UK, 1998.

[10] Lemos, R., Gacek, C. & Romanovsky, A., *Architecting Dependable Systems*, LNCS 2677, Springer, London, 2003, pp. 37-60, pp. 129-149.

[11] Leveson, N. G., *Safeware: System Safety and Computers*, Addison Wesley, Reading MA, 1995.

[12] Pumfrey, D. J., Fenelon, P., McDermid, J.A. & Nicholson, M., "Towards Integrated Safety Analysis and Design", ACM Computing Reviews, Vol. 2, No. 1, 1994, pp. 21-32.

[13] Shaw, M. & Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddle River NJ, 1996.

[14] Shrivastava, S. K. & Ezhilchelvan, P. D., "A Classification of Faults in Systems", Technical Report, University of Newcastle upon Tyne, 1989.

[15] Storey, N., *Safety-Critical Computer Systems,* Addison-Wesley, Reading MA, 1996.