Characteristics of Safety Critical Software

A. J. Kornecki, J. Erwin; Embry Riddle Aeronautical University, Daytona Beach, FL

Keywords: software, programming language, safety features

Abstract

Software in safety critical systems allows developers to implement complex functionality including safety hazards mitigation. Software may also introduce hazards by performing incorrect computation resulting in a wrong or undesired output, producing output in wrong time, or not producing it at all. The impact of safety requirements on selection of the language, design solutions, and implementation details are discussed in this paper. Certain languages tolerate developers to use error prone practices not quite appropriate for a safety-critical system. A popularity of object-oriented languages, modeling paradigm, and proliferation of Automatic Code Generation tools cause that a model can now be used as implementation conduit, rather than just analysis or design artifact.

The paper describes changing perspective on development of safety critical system with the level of abstraction moving to the early lifecycle phases from coding up to the architectural design, and increasing use of a model-based development paradigm. Such approaches improve the effectiveness of the process and promise a more cost-effective use of valuable resources. The paper will review current research into language characteristics and give some insight as to how modeling languages, particularly UML, are appropriate for system implementation.

Introduction

An Inter-Continental Ballistic Missile (ICBM) is designed to be dangerous (or willingly unsafe) to those located near the desired target where it strikes. However, it is believed to be safe to those firing it and those located at places that are not considered a desired target. A failure of ICBM hardware could make an ICBM unwillingly unsafe. Certainly, some critical component of the ICBM such as an engine, rocket structure, or the explosives could be faulty and cause destruction or an early detonation. And certainly software may fail, causing the perfect components that it was controlling capable of doing harm.

System Safety emerged as a discipline from safety concerns about the first ICBM plagued with unacceptable level of failures and accidents. Since there were no pilots for ICBM's, the accidents could not be blamed on pilot error, as was the standard for military aircraft accidents. Thus the accidents were blamed on the complexity of the systems along with the new advanced technology. Special project management structures, requiring detailed hazard analyses, and assigning responsibility for software safety were established. Proliferation of software-intensive systems moves the attention to issue of "Software Safety" [1]

By itself software is harmless. But failing, it produces (or does not produce) an output, which may set the system in an unsafe state. Software intensive system may include a failure mode where a computer generated command may lead to a hazard. Thus selection of a fault tolerant design is the essential activity for all safety critical systems [2]. For software of significant level of complexity exhaustive testing is not a viable option. Software verification process is thus usually a combination of testing, review, and analyses.

Software failure, i.e. not meeting the requirements, is attributed to errors that can originate in any of the software development lifecycle phases:

- Specification errors are due to either omission of the requirements or incorrect statement of the requirements resulting from misunderstanding of the system needs
- Design errors result from using misconceived architecture and thus preventing appropriate operation of the system. Examples of such can be lack of modularization, incomplete or incorrect interfaces, using too many or too few tasks, selecting inappropriate synchronization or communication mechanisms,

narrow path of communication or too small buffer, neglecting to apply safety mechanism and/or redundancy, allowing for a single point of failure, etc.

- Coding errors may include misspelled variable names, incorrect indices and array sizing, misplacement of synchronization primitives, initialization of variables, missing or misplaced references, endless loops, or simply syntax errors. Also there are software errors introduced by hardware, for example random electronic transients resulting in a bit flip thus changing the code of instruction/data.
- Documentation errors lead to miscommunication between different developers who rely on the artifacts from the preceding development phase. And since modern software is rather complex, any small change may have an impact on some other part of the system. An appropriate configuration management is required to reduce impact of such errors.

To address the issue of safety in software intensive systems, one needs to consider design decisions, the selection of the language, the details of the software implementation, and the potential use of modern model-based tool-supported development technologies.

Design Solutions

The typical design solutions to consider in safety critical application include: safety kernel, partitioning, monitors, watchdogs, exception handling, and variety of fault-tolerant architectures. The selection of a technique depends on the specific application from the perspective of available computing resources, time constraints, power consumption, size requirements, etc.

The concept of safety kernel has its origins in the security domain, where the security kernels have been used successfully in preventing accidental and maliciously intended access to unauthorized resources. A safety kernel keeps the unwanted behavior within the system without causing harm to the external environment. Safety kernel provides a set of mechanisms to for the detection of and recovery from safety-critical errors, and a set of policies to govern the application software's use of these mechanisms. Thus safety kernel can be treated as a passive monitor and a set of recovery routines. The correctness of safety kernel itself is sufficient to ensure the safety of the system as a whole [3]. Examples of commercially deployed systems based on the safety kernel include such as the Traffic Alert and Collision Avoidance System, and the Audio-frequency Transmission and Interlocking System, a train track signaling transmission system in Italy, cardiac pacemakers and defibrillators, etc.

Two requirements need to be satisfied for implementation of safety kernel: (a) safety properties of the system must be present at the kernel level, and (b) desired safety properties of the system will hold for all possible sequences of operations available from the kernel. An attractive aspect of safety kernel is the potential savings in the application of rigorous analysis (such as formal method analysis) to a small safety kernel in comparison to the much larger application software. Because the enforcement of safety attributes are independent of the behavior of the application software above it, less rigorous analysis can be applied to the rest of the software without affecting the safety level of the system. Since a significant burden of safety enforcement responsibilities is shifted onto the safety kernel, it helps simplify the application software constructs. Another attractive aspect is the potential for reusability. The policy enforcement of the system is centralized within a safety kernel highlighting cohesion of the safety implementation

Partitioning is a technique for providing isolation between independent system components and can be performed along the horizontal and vertical dimensions of the modular hierarchy of the software architecture. Horizontal partitioning separates major functionality into highly independent structural branches that communicate through interfaces to control modules. The control modules' primary function is to coordinate execution and communication of the functions. Vertical partitioning focuses on distributing the control and process work in a top-down hierarchy. In this hierarchy the high-level modules focus on control functions, while the low-level modules do most of the actual processing and functionality. The advantages of partitioning include simplified testing, easier maintenance, and lower propagation of side effects to other modules.

Exception handling is the interruption of normal operation to handle abnormal responses. Exceptions are signaled by error detection mechanisms to initiate fault recovery in software fault tolerant systems. There

are three main types of exception triggering events for a component: interface exceptions, internal local exceptions, and failure exceptions. Interface exceptions are triggered from an invalid service request from a component. For instance, if component *A* tries to access a non-existent function in component *B*, then an interface exception will trigger in component *B*. Local exceptions are triggered from operations within a component that are specific to that component. These exceptions are handled by the module's exception handling mechanisms. Failure exceptions are triggered when the fault tolerance mechanisms have been unable to handle a fault. This requires the calling component to determine another way to perform, or in some cases skip, the functionality in the faulty component.

Software fault tolerance is typically used as an extra defense against possible faults by enabling the continued delivery of services at an acceptable level of performance and safety after a design fault becomes active. Techniques for software fault tolerance can be divided into two groups. Single version programming techniques focus on improving the design of the software by detection, containment, and handling of errors caused by the activation of design faults. Multi-version techniques use multiple versions of software component to ensure that design fault in one version does not cause system failure. Both techniques can be applied at functions, processes, and the entire system level. The techniques may be applied only to the components most likely to contain software faults, resulting in less time spent developing the software system and more time focusing on the important safety aspects of the design.

In those situations where a fault will appear for no visible reason, create the error, and then go away, the most common single-version fault tolerance technique is checkpoint and restart. It provides a way to restart the component while being independent of any damage caused by a fault. This technique is popular in resource-constraint systems. For systems that can afford it, multi- versions of software modules can be executed in either series or parallel. Each version has a separate error detection mechanism and acceptance test. Several different voting techniques can be used to detect discrepancy between versions. The rationale for using multiple versions is that different engineers build their respective versions differently. As a result, each version is expected to fail differently since software faults are inherent to the design of software. If one version fails, then one of the alternate versions should be able to provide full recovery and operation. The approach, although used widely in safety critical aviation and airspace systems, has been contested; the argument being that the source of common errors may be in common requirements. Recovery Blocks (RB) and N-Version Programming (NVP) are the most popular techniques in this category [4].

Recovery blocks combine the single-version technique of check and recovery with the context of multiple versions of a software component. The alternate component is tried after the primary component fails an acceptance test. If all alternate versions have been tried unsuccessfully, then the original component raises an exception. The success of the technique depends heavily on quality of the application-dependent acceptance test. For N-Version programming the decision for output is based on a comparison of all the version outputs. A majority voter mechanism selects the final output thus acceptance testing is not required. Several other techniques like N-Self-Checking Programming or Consensus Recovery Blocks are combination of the basic the RB and NVP. A judicious application of appropriate voting mechanism and acceptance test is the critical consideration for selection of specific technique.

Implementation

The specific techniques recommended for all safety-critical systems and required for the systems of the highest level of assurance vary, depending on the specific development phase.

The following techniques assist to reduce safety risks in the requirements phase: flow down of safety requirements from system design and system hazard analysis, software safety requirements analysis, checklists and cross-references, software criticality analysis, formal inspection, timing analysis, sizing analysis, and throughput analysis. At the architectural and design phase it is imperative to employ such techniques as hazard risk assessment, architectural design analysis, independent analysis, software fault tree analysis, state modeling, design data and interface analysis, constraint analysis, complexity assessment, and formal inspections. At the implementation and coding stage the developers need to consider: safe programming subset, code standard checklists, defensive programming, logic analysis, data analysis, interface analysis, unused code analysis, and formal inspections. Testing should consider boundary values,

coverage analysis, operational modes, paths, statements/branches/predicates, loops in varying scenarios, timing constraints, data location in I/O space, arithmetic accuracy, modules in their environment, and performance monitoring. Judicious applications of these techniques will not only facilitate development of safe(r) system but also provide rational arguments about the system safety for certification purpose.

Using formal notation facilitates creation of requirements that are more complete and less unambiguous. These requirements can be validated using formal method techniques or a model verification tools checking behavioral finite state model of a system against expected properties of that system. If the specification can be executed, then it must be run on an appropriate set of inputs to look for expected behavior.

Runtime checking methods include self-checking code, or the development of an independent monitor. Self-checking code include additional checks inserted by either the developer or compiler. However, it may add considerable time to the execution of the system. The development of an independent monitor is useful, because it acts independently of the software and checks the outputs from the software or the state of the entire system.

Language Selection

The selection of a programming language may impact the programming errors ranging from mistyping variable name to misunderstanding and thus incorrectly encoding an algorithm. The expressiveness and style of selected programming language may impact the programmer's ability to avoid mistakes. Specific language may help programmer to deal with failures or unexpected inputs, and to structure and test whole programs and modules. No language is ideal in all these respects. Syntax of some languages may be so convoluted that the code can be misunderstood. An example of such potential misunderstanding may be the rules for operators' precedence. Different compiler writers may interpret ambiguous features of language in a different way, resulting in diverse behavior of apparently the same code segment. Hatton [5] quotes 195 items of the C language that the ISO standard committee never agreed upon, leaving the interpretation to the compiler writers.

U.S. Government mandated the use of Ada, a block-structured programming language, for real-time software systems for Department of Defense (DoD). Ada provides asynchronous transfer of control mechanism based on the termination model feature, supports explicit exception declaration, and propagation of not handled exceptions. It supports concurrency on the language level with Object Oriented Programming (OOP) concepts like inheritance, automatic object initialization, and run-time dispatching of operations through tagged types and dynamic polymorphism. It does require run-time platform for the execution of its programs. Ada 95 provides high level of synchronization with barriers concept provided by mutual exclusion of protected objects. It has a built real-time timer and calendar for accurate timing of task [6]. Ada95 has been used in the software development of CH-53E Naval Helicopter. The developers of Boeing 777 software systems have enjoyed the Ada's portability, code reuse features, built-in safety features that helps reduce development time, expense, and concern for debugging the software. An automatic train control system TVM 430 was successfully developed using Ada because of the language maintainability, portability, and strong typing – ideal for safety applications [7].

The use of synchronous language Lustre is widely known in the development of safety critical real-time systems [8]. The synchronous assumption could be an oversimplification, inconsistent with the real-time systems [9], but tolerated due to its impact on the verifiability of the application. Based on Lustre, SCADE Suite tool (Safety Critical Applications Development Environment) captures formal specification that can be immediately executed through simulation while allowing for formal checking of safety properties [10]. This helps to capture specification errors and modify them before automatic generation of the code. It generates safety critical software and considerably reduces development cost. SCADE Suite has been used to create software for autopilots, flight and engine control, braking, cockpit display, power management, etc.

There are three main sources of errors in a language resulting from the language constructs [5]:

- The standards committee could not agree thus purposefully leaving an ambiguity in the standard.
- The language constructs were mistakenly left undefined.

- The language constructs were well defined and agreed upon, but too complex and/or confusing thus either directly or indirectly contributing to developer errors.

For example, the C language current standard maintained by ISO has 195 items currently in the first category. These are things that the standard recognizes are not defined, and are left up to a compiler implementer to decide [5]. Programmers may misinterpret the effect of constructs in a language. There are quite a number of areas of the C language that are easily misunderstood by programmers. For example, the rules for operator precedence are well defined but very complicated. It is easy to make wrong assumptions about the sequence of computation in a complex expression.

If a language features are not completely defined or ambiguous, a programmer can assume one thing about the meaning of a construct, while the compiler can interpret it differently. A language compiler is itself a software tool. Compilers may not comply with the language standard in certain situations, or they may simply contain 'bugs'. Compiler writers have been known to misinterpret the standard and implement it incorrectly. In addition, compiler writers sometimes consciously choose to vary from the standard.

A somewhat different language issue arises with code that has compiled correctly, but causes errors in the running of the code due to the particularities of the data. Languages can build run-time checks into the executable code to detect many such errors and take appropriate action. C is generally poor in providing run-time checking. This is one of the reasons why the code generated by C tends to be small and efficient, but there is a price to pay in terms of detecting errors during execution. C compilers generally do not provide run-time checking for such common problems as arithmetic exceptions (e.g. divide by zero), overflow, validity of addresses for pointers, or array bound errors.

Language Requirements

Table 1 presents basic language characteristics to be considered for safety-critical application. A selection of language requires analysis of these characteristics and evaluation of their impact for a specific project. A rating may be given to each characteristic and the resulting totals would help to make decision regarding the technical aspects of language selection.

Table 1 – Language Characteristics

Language Characteristics	**Rationale**
Strong typing	Help reduce errors in programs at compile-time, enhances the integrity and security
No side effects	Prevent programs to behave in an ambiguous, or possibly unpredictable way
Modularity and structure	Assure that the complexity of software becomes manageable
Formal semantics	Help to produce quality software, often cost-effectively
Well-understood semantics and syntax	Easy to adopt and to implement
Domain specific	Include robust mechanisms for controlling memory, I/O devices or other hardware
Concurrency	Language-level support for multitasking or multithreading, control over scheduling policy and straightforward communication and synchronization mechanism
Predictability	Functional and temporal behavior can be predicted
Run-time environments	Trusted/certified translators are used
Handling errors	Graceful degradation and recovery mechanism outweigh overheads and possible unpredictable behaviors
Model of mathematics	Integer and floating point arithmetic
User documentation	Improve program readability and maintainability
Enumeration types	Help reduce errors
Coding style	Reduce gap between well-established software engineering principles and the actual practice of programming

Abstraction or information hiding	Decrease software complexity and support modularization
User specified assertions	Supports analyses capability and design by contract thus defect reduction
Expressive power	Ability to solve problems and implement all algorithms
Language subset	Enforce the rules/characteristics and simplify the code
Certified analysis tools	Support development, check for errors, such as race conditions and deadlocks
Interface	Facilitates development
Code initialization	Improve the efficiency of programs
Portability	Reduce analyses when porting to another platform

It needs to be noted that language should not simply be evaluated on how "safe" it is, but how safe it can be made. Tools and documentation of language restrictions may transform an "unsafe" language into a "safe" language. A well-defined language subset may identify which elements of a language should be used to maximize the aforementioned safety characteristics of the language. Just as the language itself, the subsets should be reviewed and approved or certified to be of any real value. Using a well-defined subset is often the best choice since it will allow a development team to use a well known widely used language; leverage existing skills, knowledge base and tool support. Two of the most widely used and accepted subsets are the Ravenscar Profile for Ada and MISRA C.

The Ravenscar Profile is formally defined in terms of Ada95 constructs, and this definition has been accepted for inclusion in the revision to the ISO standard definition of the Ada language that is scheduled for 2005 release. The full definition is contained in a guide on using the Ravenscar Profile for high integrity systems. The main components of the Ravenscar profile call for [11]: a fixed set of threads, a fixed set of protected objects that provide mutually exclusive access to shared data, a fixed set of synchronization objects, a fixed set of interrupt handlers, a synchronous delay facility based on absolute time values, a deterministic fixed-priority preemptive thread scheduling policy, and a policy to enforce mutual exclusion.

Having carried out extensive consultation within the automotive industry, the MISRA consortium has now completed the development of guidelines specifically aimed at the use of the C language. These guidelines primarily identify those aspects of the C language, which should be avoided in safety-related systems, along with other recommendations on how other features of the language should be used. It is anticipated that the guidelines will be adopted for embedded C programming throughout the automotive industry [12].

C was chosen for use of the "safe" MISRA subset due to great flexibility, mixing of high level abstraction and low level access, memory management techniques, wide usage, excellent tool support and clear understanding of its shortcomings. C is an excellent example of how it can be advantageous to modify a popular general-purpose language to be used in a safety critical domain.

Modeling Languages as Implementation Languages

Model-based architectural approach has been proposed for improving predictability of real-time embedded system utilizing automated analysis of tasks and communication architectures [13]. The emerging trend is that of model-based development allowing an automatic generation of run-time executive responsible for task dispatching and inter-task communication. Originally, MetaH language has been used to support this approach [14]. It has been pre-cursor to emerging standards in the Society of Automotive Engineers (SAE), Avionics System Division (ASD), working group on Avionics Architecture Description Language (AADL), and Object Management Group (OMG) continuous extensions to the UML notation. The stress is on the fact that modern software intensive systems have short lifecycle span with frequently changing technology and operational environments. The new version of development tools, compilers, operating systems, and hardware platforms render 3-5 years old systems difficult to maintain and modify. It is particularly visible for embedded systems constituting the core of safety-critical domain.

Since UML grew out of a need to support Object Oriented Design and Analysis, UML is very effective at information hiding, encapsulation, and modeling reusable objects. It has strong type checking and a written formal semantic. The UML paradigm support for characteristics listed above is still under discussion. The general consensus is that, despite current advances on establishing specialized UML profiles, UML is not quite "formal" enough for use as a language definition [15]. The UML paradigm is supported by a variety of well-established tools used widely in developing enterprise software. Many of the tools have code generation capability which moves the level of abstraction upwards focusing developer on the design problem as opposed to mundane code hacking.

The argument as to whether we can trust modeling tools that "automatically" generate the source code is simply a rewording of the old question as to whether we can trust compilers to transform high level source code to machine language. Compilers are now trusted tools and essential if we wish to program in a high level and language and gain the benefits of this higher level of abstraction. Similarly, to gain the benefits of using a modeling language as an implementation language, one have to reach the same level of confidence with tools supporting that language. There are several challenges to prevail before such level of confidence can be reached.

The following quote from the FAA expresses the idea that modeling languages can be useful, while raising an important concern:

> "When using object-oriented (OO) tools to develop software requirements, design and implementation, it is beneficial to work at the visual model level, especially when using UML. When working with OO tools, configuration management might be done at the modeling level (i.e., diagrams). This may cause a concern when the OO tools can introduce subtle errors into the diagrams." [16]

An experiment was designed to highlight the issue of the abovementioned concern. The experiment consisted of creating a relatively simple state machine (Figure 1), and then implementing it using three tools with code generation capability. The tools allow developer to simulate the system behavior and the events. Would the same state machine implementation result in different behavior?

The selected tools were Statemate v3.2, Rhapsody v4.01 for C, and Rhapsody v4.01 for Java (all by iLogix, the latter two being the same tool, but with a different implementation language). The rationale for tool selection was: availability, familiarity with the tool, support for state machine representation, and the commonality of the vendor.

The state machine in Figure 1 was drawn in Statemate. One half of the concurrent state machine, SUBCHART2, generates the events, and the other half, SUBCHART1, consumes them. One action in the state machine that is not readily apparent is an action upon entry inside S1, marked by > next to the state name. This action consists of setting the variable A to TRUE. The UML specification written by the OMG states [17]:

> "Whenever a state is entered, it executes its entry action *before* any other action is executed."

When SUBCHART1 transitions to state S1, A will become TRUE immediately, thus no other events should be consumed, nor transitions take place. When SUBCHART2 transitions from START_STATE to GO_BACK, EV1 is generated, and A is set to FALSE. The specification defines the following actions that occur during transitions:

> "The processing of a single event by a state machine is known as an *run-to-completion step*. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all actions (but not necessarily activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event will never be processed while the state machine is in some intermediate and inconsistent situation. The *run-to-completion* step *is* the passage between two state configurations of the state machine."[17]

The specification also states that the actions of any given transition should be completed in the order they are listed. In this case, the event should be generated and then A = FALSE. SUBCHART1 will start out in S1 setting A = TRUE. SUBCHART2 start out in START_STATE. Upon the first cycle of execution the guarded transition will immediately fire generating EV1, which should then send SUBCHART1 to state S3 as part of the run-to-completion of the EV1 generation. After this A should be set to FALSE. Once in S3

SUBCHART1 will wait for EV2 to be fired and then return to S1 where A = TRUE. SUBCHART1 should never reach S2, since A is always reset to TRUE upon entry to S1. That is the behavior that was expected. Results of the experiment are reflected in Table 2.

Table 2 – Experiment Results

Tool	**Behavior**
Statemate	Upon receiving and consuming EV1, the SUBCHART1 alternated between entering S2 and S3. The specific timing of Statemate in resetting A to TRUE allowed the entry to state S2, even though it was not intention of the original design.
Rhapsody for C	With the guarded transitions between states in the SUBCHART1, transitions occurred between states in the SUBCHART1 only, and the events are generated, but EV1 and EV2 where never consumed. No entry into S2 or S3 ever occurs. *NOTE: A second state machine with timed transitions between START_STATE and GO_BACK in SUBCHART2 was created to test that the tool was being correctly used to generate and consume events. With timed transitions between the states the events EV1 and EV2 were consumed regularly as they were generated. However only S3 was entered, S2 was never entered.*
Rhapsody for Java	With the guarded transitions occurring and the generation of events EV1 and EV2, the events EV1 and EV2 would occasionally be consumed. Several transitions would occur between START_STATE and GO_BACK before either EV1 or EV2 would be consumed. Again, state S2 was never reached, only S3.

The behavior of the system in the three implementations was different. Statemate varied from the expected desired behavior, causing entering to state S2. Statemate does not accept implementing the UML standard for state machines. Instead the documentation defines how state machines behave. The OMG actually references these differences in the UML standard as well, listing differences between UML state machines and classic or Harel statecharts. Eleven differences are listed, but only one seems to apply in this case:

> "Classical statecharts are based on the zero-time assumption, meaning transitions take zero time to execute. The whole system execution is based on synchronous steps where each step produces new events that will be processed at the next step. In object-oriented state machines, these assumptions are relaxed and replaced with these of software execution model, based on threads of execution and that execution of actions may take time."[17]

Supposing that in Statemate the transition takes zero time [18], event EV1 would fire, A=FALSE would happen simultaneously (at least in the logical sense). Although Statemate still supports the run-to-completion idea that UML state machines support, which is *not* a difference between the two. How then to reconcile the fact that EV1 should run to completion and fire a transition, yet A=FALSE at the same moment? Statemate seems to reconcile it by advancing one transition treating A as FALSE and the next time treating A as TRUE.

Rhapsody takes a different approach than Statemate to state machines and does comply with the UML 1.3 standard. It creates an event queue and then dequeues the messages for consumption [19]. The underlying runtime model seems to be culprit for the behavioral discrepancy between Rhapsody for C and Rhapsody for Java. The Java model allowed the SUBCHART1 some time to process and consume the events while the C runtime did not. The Java model, however, was far from deterministic and it was not consistent in regard to when the events would be consumed.

The point of this experiment was not to show errors in the tools or claim that the behavior observed was incorrect. It is simply to point out that problems could arise if a developer familiar with UML state machines wished to implement them with UML while using an automatic code generation tool. How can the designer of the behavior be assured that the desired behavior will be in the resultant system if the tool does not implement the diagram in the same manner as the designer intends it to be implemented?

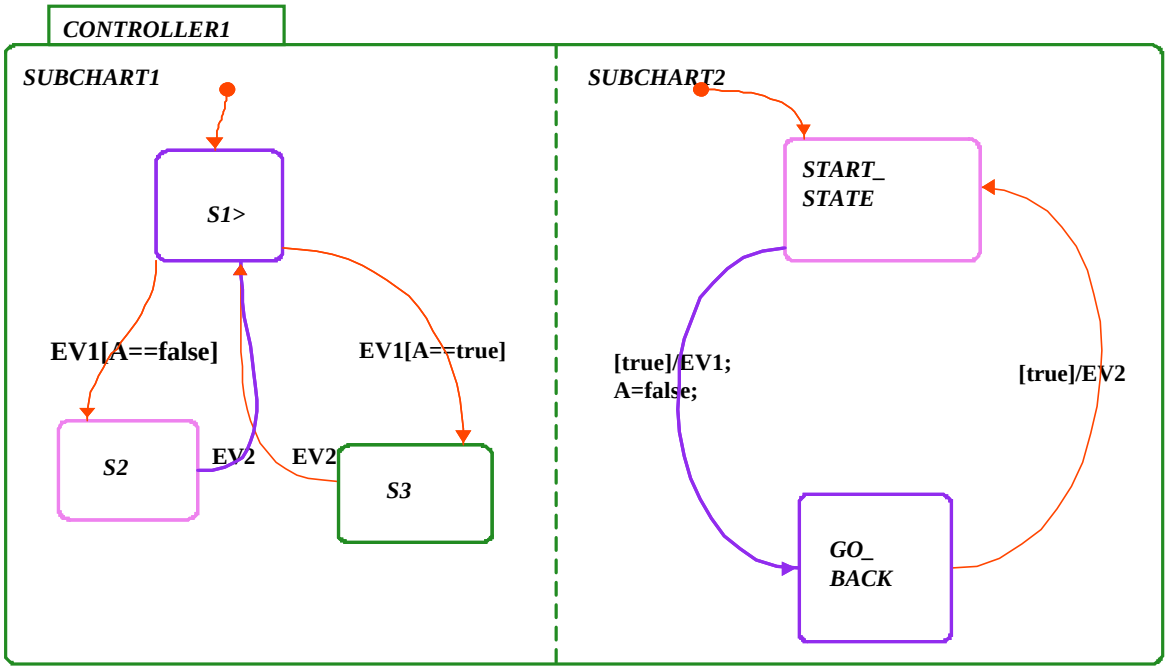

Figure 1 – Test Case Design

Conclusion

The choice of a language can have a significant impact on the success or failure of a safety-critical system. The language can impact the ease of validation, the number of defects, and many important parts of the development process. Few languages are inherently "safe" as well as having good tool support, good documentation and wide usage. A general-purpose language, which is made "safe" by use of a subset and good tool support, is the best choice for a safety-critical system. Modeling languages show excellent promise as implementation languages for all types of software development, not just safety critical. The graphical notation of modeling languages facilitates communication, allows for a much greater range of abstraction, and allows developers to focus on important engineering concepts rather than implementation details. UML has excellent descriptive capabilities and is it possible to model a wide range of systems effectively in UML. UML also supports many of the safety language characteristics (such as strong typing). In regards to safety-critical systems and hard real-time systems UML seems to be lacking in certain areas. Certain ambiguities in the language must be addressed, one example being shown in the simple experiment in this paper. Also more support for performance and timing needs to be included in tools and adopted by users. Perhaps, as with the most popular programming languages, a "safe" subset of UML needs to be defined. Such development techniques as checking requirements and design through review and analysis techniques find and mitigate mistakes early in the development process. Also, the use of formal methods to define requirements and design supports creation of unambiguous documentation. During the implementation phase, added code reviews and self-checking code discover hidden bugs in the system not found during the earlier reviews and inspections. These are all important to developing software at a safety-critical level.

References

1. N.Leveson, Safeware: System Safety and Computers, Addison Wesley, 1995.
2. K.C.Kang, P.R.H.Place, Safety-Critical Software Status Report and Annoted Bibliography, CMU/SEI-92-TR-5, June, 1993
3. John Rushby, Kernels for Safety? Safe and Secure Computing Systems, Ch.13, pages 210-220. Blackwell Scientific Publications, 1989
4. L.Pullum, Software Fault Tolerance: Techniques and Implementation, Artech House, 2001

5. L.Hatton, Safer C, McGraw-Hill 1995
6. A.Burns, A. Wellings, Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX, Addison Wesley, 2001
7. R.daCosta, "The History of Ada", Defense Science March 1984
8. P.Caspi, et al, Formal Design of Distributed Control Systems with Lustre, SAFECOMP99, LNCS 1698, Springer-Verlag, Berlin, 1999, pp. 396-409
9. K.M.Kavi, Real-Time Systems Abstractions, Languages, and Design Methodologies, IEEE Computer Society Press Los Alamitos, CA, 1992
10. F.X.Dormoy, "SCADE: a Toolbox for the Development of Safety Critical Control Systems", presentation at Critical Systems and Software Seminar, JSLC 2001, Grenoble, November 2001,
11. D.Dobbing, "The Ravenscar Profile for Real-Time and High Integrity Systems", Crosstalk, Nov 2003
12. MISRA, "Guidelines For The Use Of The C Language In Vehicle Based Software", 1998
13. P.Feiler, B.Lewis, S.Vestal, Improving Predictability in Embedded Real-Time Systems, Report CMU/SEI-2000-SR-011, Pittsburgh, PA, 2000B.Selic, "Object Oriented Modeling and Safety Critical Software", Rational Software Group, IBM, Workshop on Critical Systems Development with UML, Oct 2003
14. S.Vestal, "Formalizing Avionics Software Architectures", presentation at Critical Systems and Software Seminar, JSLC 2001, Grenoble, November 2001
15. Z.Pap, I. Majzik, and A. Pataricza, "Checking General Safety Criteria on UML Statecharts", Computer Safety, Reliability and Security, Proc. 20th Int. Conf., SAFECOMP-2001, Springer, 2001, vol. 2187 of LNCS, pp. 46–55
16. FAA, "Issues and Comments about Object Oriented Technology in Aviation", January 29, 2004 http://shemesh.larc.nasa.gov/foot/ootcissues.pdf
17. Object Modeling Group, "OMG Unified Modeling Language Specification" version 1.3 November 1999
18. D.Harel, M. Politi, "Modeling Reactive Systesm with Statecharts: The Statemate approach", iLogix 1999
19. I-Logix, "Rhapsody User Guide", iLogix 1997

Biography

Andrew J. Kornecki, Ph.D., Professor in the Department of Computer and Software Engineering at Embry Riddle Aeronautical University, 600 S. Clyde Morris, Daytona Beach, FL 32114. E-mail: andrew.kornecki@erau.edu, (386) 266-6888.

Dr. Kornecki's research and teaching interest include: modeling and simulation, real-time systems, performance analysis, and software safety with the results published in journals and conference proceedings. He served as a Visiting Scientist at the Federal Aviation Administration and National Academy of Sciences Committee on Aging Avionics in Military Aircraft. Recently he has been engaged in FAA sponsored research on testing and certification and assessment of development tools for safety critical real-time systems.

Jared Erwin, BSCS, Master of Software Engineering Program at Embry Riddle Aeronautical University, 600 S. Clyde Morris, Daytona Beach, FL 32114. E-mail: jared.erwin@erau.edu

Mr. Erwin has undergraduate degree from Westminster College in Salt Lake City, UT. He is pursuing a Master's Degree in Software Engineering. His current research focus on model centered software development in safety-critical domains. Jared is an IEEE student member, ACM student member and member of UPE. He has been offered position with Software Engineering & Independent Requirements Verification at Cardiac Rhytm Management, Guidant Corporation, St.Paul, MN.