09ATC-0271

Challenges in Validating Safety-Critical Embedded Systems

Peter H. Feiler
Software Engineering Institute

ABSTRACT

The embedded software has played an increasing role in safety-critical systems. At the same time the current development process of “build, then integrate” has proven unaffordable for the Aerospace industry. This paper outlines challenges in safety-critical embedded systems in addressing system-level faults that are currently discovered late in the development life cycle. We then discuss an architecture-centric approach to model-based engineering, i.e., to complement the validation of systems with analysis of different operational quality aspects from an architecture model. A key technology in this approach is the Architecture Analysis & Design Language (AADL), an SAE International standard for embedded software system. It supports analysis of operational qualities such as responsiveness, safety-criticality, security, and reliability through model annotations. A number of industry initiatives have been underway to demonstrate the feasibility of using this technology in industrial practice.

INTRODUCTION

Embedded software systems have become a key component of systems, both in terms of contribution to the safety-criticality of the system and in terms of cost. For example, in the Aerospace industry the size of the onboard software has doubled every four years and has reached over 20M sources lines. Unfortunately, current software development processes of build, integrate and test are reaching their limit of affordability, as the cost of repair has grown. A 2002 National Institute of Standards and Technology (NIST) study [1] shows that 70% of faults to be introduced early in the life cycle, while 80% of them are not caught until integration test or later with a repair cost of 10x or higher. Figure 1 shows percentages or fault introduction, discovery, and cost factor. If we can discover a portion of those faults earlier in the life cycle, we have the potential of leveraged cost savings.

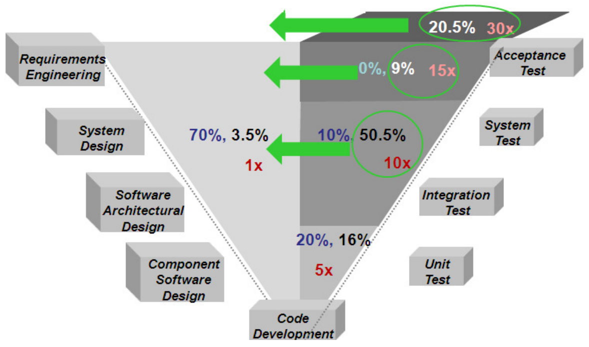

Figure 1: Benefits of Early Fault Discovery

In this paper, we examine the faults that persist late into the development life cycle. In particular, we ask why it is the case that, despite that fact that these systems have been designed with fault tolerance and safety-criticality in mind, a high percentage of system-level faults are discovered late. This will provide us insight into the potential root cause areas and provide guidance for solutions.

Model-based engineering is considered to be a key to improving system engineering and embedded software system engineering. Modeling, analysis and simulation has been practiced by engineers for a number of years. For example, computer hardware models have been created in Very High Speed Integrated Circuits Hardware Description Language (VHDL) [2] and validated through model checking [3]. Control engineers have used modeling languages such as Simulink for years to represent the physical characteristics of the system to be controlled and the behavior of the control system.

Characteristics of physical system components, such as thermal properties, fluid dynamics, and mechanics, have been modeled and simulated. Even for software systems application models have been created by modeling languages with limited formalism, such as Unified Modeling Language (UML), and more formal analytical models have become the basis for resource and timing analysis, fault impact and reliability analysis, safety-criticality, and security analysis. Despite these modeling efforts system-level problems remain late into the development life cycle due to inconsistency between the analytical models and with respect to the evolving system being modeled (Figure 2).

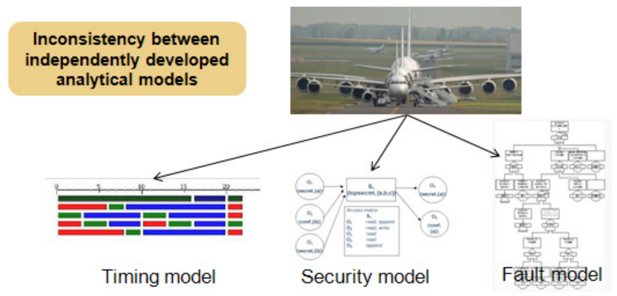
Inconsistency between independently developed analytical models
Timing model
Security model
Fault model

Figure 2: Inconsistent Analysis Truth

We will first examine different types of system-level failures that have occurred in actual systems and are related to embedded software. Next, we identify root cause areas for these system-level faults and suggest SAE AADL [4] as a notation to capture the relevant semantics in an architecture model. Finally, we discuss an architecture-centric approach to validation of embedded software systems that shows promise and is embraced by the Aerospace industry.

SYSTEM-LEVEL FAILURES

Why do system-level failures still occur despite the deployment of fault-tolerance techniques in systems? System engineering approaches in the form of hazard analysis consider the system in its operational environment to address safety-criticality concerns of key capabilities. Similarly, fault tolerance approaches such as hardware redundancy for managing hardware failures are well established in both system engineering and computer hardware engineering.

There may still be mismatched assumptions made between different members of an engineering team. For example, the control engineer may make assumptions about characteristics of physical components, such as lag, when designing their control algorithm. A system engineer may make assumptions about heat dissipation of the computer board designed by the computer engineer. These assumptions must be documented and validated throughout the life cycle. Software engineers and application developers are often not involved until later in the life cycle to implement the desired functionality. Especially in embedded systems, decisions made for the runtime architecture and its implementation can have strong impact on the system operation as they may violate assumptions made by system and control engineers. Therefore, it becomes imperative to understand this impact earlier in the life cycle by identifying assumption mismatches.

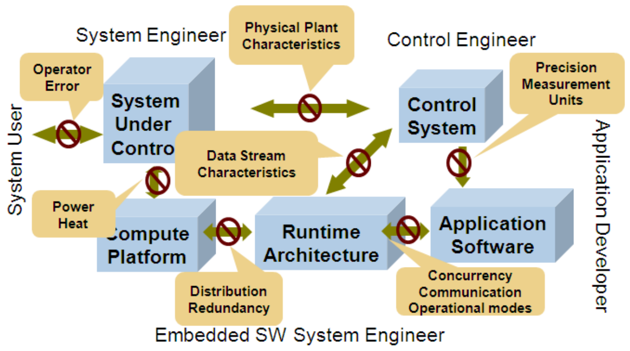

Figure 3: Mismatched Assumptions

Figure 3 illustrates different dimensions of mismatched assumptions. For example, a hardware engineer may upgrade a processor to accommodate increased processor demand by the application. This may result in a change in weight, electrical power consumption, and heat dissipation. A control engineer designs a control algorithm for an autopilot under the assumption that it is used on a larger plane with a lag in physical responsiveness and chooses a processing rate accordingly. Deployment of this autopilot in an aircraft with different response characteristics may result in uncontrolled flight behavior. Maintenance operations must be included in the safety cases to be validated as was illustrated by the Maglev accident between a train and a track maintenance vehicle [5].

An example of mismatched assumptions in the translation of functionality into application software is the explosion of the Ariane 5 rocket during her maiden flight. The destruction was triggered by the overflow of a 16-bit signed integer variable due to Arrina 5's greater acceleration in a reused Ariane 4 software component to perform a function that was "not required for Ariane 5" [6]. The reason for the overflow was the representation of a vertical velocity value as a 16-bit integer—placing a restriction on the maximum value. Ariane 5 flight path caused the value to overflow. This fault was not caught because the handler was disabled due to efficiency considerations and cascaded into a total system failure.

A mismatch in measurement units between gallons and liters in the fueling process of aircraft lead to an emergency landing [7]. This occurred during the time Canada was converting to the metric system. The aircraft fuel management system of this aircraft was already operating in kilograms (kg) instead of pounds (lbs). When a fault in the dual redundant fuel quantity indicator system occurred, which resulted in a blank display of fuel gauges, the pilot had to perform manual conversion, but used the conversion factor from liter to lbs instead of kg, before entering the data into the system.

After years of development, F/A-22 flight tests began in late 1997, but the aircraft still experienced serious avionics instability problems as late as 2003. According to testimony from the U.S. General Accounting Office (GAO), “The Air Force told us avionics have failed or shut down during numerous tests of F/A-22 aircraft due to software problems. The shutdowns have occurred when the pilot attempts to use the radar, communication, navigation, identification, and electronic warfare systems concurrently.” [8]

When laptops with dual core processor came out ITunes crashed randomly when ripping a music CD [9]. ITunes was designed as multi-threaded application, with one thread determining the dB level of tracks, while the second thread converting the audio. A single processor system executed first one task, then the second task. On a dual core processor, the two concurrently executing threads were attempting to update the same music catalogue without explicit synchronization. In other words, the original implementation had a mutual exclusion requirement that was not implemented, assuming sequential execution of tasks.

Mutual exclusion is often implemented by locking data structures as critical regions. If a fixed-priority pre-emptive scheduling protocol, e.g., Rate-Monotonic Scheduler (RMS), is used this can lead to unbounded priority inversion. The Mars Pathfinder had a classic case of priority inversion [10]. Once recognized, the priority ceiling protocol (PCP) [11] was enabled on the locking semaphore and the problem resolved. This illustrates the value of formal analysis, which could have been performed on this system well before its launch. However, PCP implementations make the assumption, in that it assumes that tasks are scheduled by a single processor. If we migrate an application synchronized with PCP to a dual-core processor, the original protocol will not guarantee mutual exclusion – a distributed priority ceiling protocol (DPCP) must be used instead.

When system components become virtualized assumptions about physical redundancy will be violated. In 1986, the Internet, then ARPA-net, was accidentally split into two networks [12]. All seven trunk lines to New England, which used to be separate physical lines, were severed when AT&T suffered a fiber optic cable break, which lasted 11 hours. When AT&T had gone fiber optic, these physically redundant trunk lines became logical trunk lines on this much higher bandwidth connection, losing all physical redundancy. The same occurs, when you replace your primary and backup 100GB hard-drives in your desktop with a 500GB drive and configure it with a primary and a backup partition. Your backup software will operate, but your backup copy is lost on a hard-drive crash.

The ARINC 653 standard [13] uses virtual machines for time and space partitioning to isolate application subsystems from affecting each other due failures. Commercial real-time operating systems (RTOSs) support ARINC 653 and it are used in avionics systems. Assumptions made by application software, when running on dedicated hardware, do not always behave the same when the same software operates in such a virtual machine. For example, in an early test in the lab the implementation of a partitioned architecture based on ARINC 653 for an aircraft executed at 2/3 speed. Upon closer inspection, unanticipated resource contention was identified as the culprit. One application partition initiated a Direct Memory Access (DMA) transfer just before its timeslot was over. Another partition was scheduled, but the DMA transfer of the original partition continued. The DMA transfer utilized a bus that was also used by the operating system to swap the cache content of the processor when switching between partitions, thus, slowing the partition switch. In addition, the DMA transfer accessed a memory bank that also contained the application code of the second partition, causing slow instruction fetches.

Partitions introduce virtualization of time, while the application code may assume actual processor time. For example, control system application periodically samples a data stream at the beginning of each frame, when executing on a dedicated processor with a static scheduling scheme. However, when executing in a partition, the same application code samples the data stream at the beginning of the partition timeslot, which may be offset from the frame.

In the 1990’s the flight software for a fighter was migrated to an Integrated Modular Avionics (IMA) architecture. The application software was originally implemented as a cyclic executive with periodic sampling tasks. When ported to a RMS, the display showing tracked objects randomly blurred. The transfer of target data from the sensor to the display, which predictably took four frames in the original system, now varied between four and eight frames due to pre-emption and depending on the workload. This showed itself as an oscillating target symbol of the tracked object [14].

When such data is sampled by a control algorithm, this sampling jitter can results in instability of the controller [14]. In other words, the time-sensitivity of the data stream is affected by a change in the scheduling protocol and the use of a non-deterministic communication scheme (sampling of shared variables). This manifests itself to the control system as noisier sensor data, which must be compensated for with more complex algorithms and calibration.

In the late 1990’s an attempt at a well-intentioned performance improvement of ground station software that tracks objects close to a space craft had unplanned side effects. The subsystem collecting the tracking information had originally sent a complete map of tracked objects to the command and control subsystem. In order to reduce the load on the network, a change was made to communicate only changes to the map. Unfortunately, this communication occurred over a network protocol that drops packets under overload conditions. As result, during integration testing, it was discovered that state changes randomly were not delivered. In other words, the data representation of the communicated data stream assumed guaranteed delivery.

In 2008, a Qantas flight unexpectedly dropped up to 650 feet multiple times within a few minutes [16]. A fault in one of three Air Data Inertial Reference Units (ADIRU) caused the unit to supply incorrect data to other aircraft systems and led to automatic disengagement of the autopilot, false stall warnings, and loss of attitude information on the pilot display. With the autopilot off, two minutes later the primary flight control computer still received false data from the ADIRU and commanded a major pitch down. A failure in one component of a triple redundant unit caused an operational mode change and operational response to a data stream by another subsystem without recognizing its faulty nature, i.e., assuming a correct data stream due to the redundant nature of the source.

In the next section we will summarize root cause areas that contribute to these system-level failures and introduce AADL as a notation to capture relevant aspects of systems to address these root cause areas.

ANALYSIS OF ROOT CAUSES

System-level faults identified in the previous section fall into several root cause areas and typically are related to undocumented assumptions that are not validated as a system evolves throughout the life cycle.

One root cause area is the flow of information through the system. As multiple components are involved in its handling, all can affect its characteristics and can be affected by such changes. Therefore, it is essential to document assumptions made about such data streams. These assumptions fall into several categories:

- Assumptions about the data of a data stream: this includes the application data type, e.g., temperature, representation of state or state change, its base type representation, e.g., 16 bit unsigned integer, acceptable range of values, base value that is represented as zero, e.g., -50, and measurement unit, e.g., Celsius.
- Assumptions about the timing of the data stream: age of the data, e.g., time since it was read by a sensor, data latency, i.e., handling time of new data, and latency jitter, i.e., variation in latency. Contributors to age, latency, and latency jitter of data streams, both in terms of application logic and computer platform.
- Assumptions about the stream characteristics: data stream completeness and acceptable miss rates, acceptable limits in value changes between elements of the data stream.

A second root cause area deals with performance impact. In integrated modular avionics (IMA) architectures the computer resources as well as physical resources are shared and concurrent use can lead to resource contention. Therefore, assumptions about availability of resources and resource guarantees must be documented. These assumptions fall into several categories:

- Undocumented resource sharing: record of all users of a resource - direct and indirect, logical and physical; accountability for peek demands; assurance of mutually exclusive use requirement.
- Impedance mismatch of resource demand and capacity: demand may exceed capacity, or capacity of one resource may exceed capacity of connected resource. For example, a high bandwidth Gigabit Ethernet network can flood low-powered processors resulting in denial of service and lower than expected processor speed.
- Unmanaged hardware resources: individual high demand component may dominate an unmanaged resource, e.g., high-volume traffic by one transfer can cause delay and denial of transmission service on an unscheduled network. Enforcement of resource budget limits is essential to safety criticality.

A third root cause area is virtualization of processor, network, and memory resources. Such virtual resources that represent logical resource capability and capacity that can be allocated to physical resources in different configurations. System architectures utilize these virtual resources and assume certain guarantees.

- Resource isolation guarantees: In addition to logical resources, limit enforcement virtual resource concepts of processor partitioning (ARINC 653) and virtual channels represent information access and flow boundaries. Enforcement of such assumed isolation regions must be validated in the context of time-shared resources.
- Virtualization and redundancy: Virtualization turns physical redundancy into logical redundancy. Deployment allocations must be taken into account to ensure assumed reliability and availability.
- Virtualization of time: virtualization of computer platform processing time, and of time servers. Tasks and partitions virtualize the time of application code execution. Time-sensitive application interactions are affected by time synchronization across computer platform components, i.e., its operation in a synchronous system (operating with one clock) or as globally asynchronous system. Applications processing time-sensitive data, e.g., environmental observations for a common operational picture (COP) use time stamping. A common time reference is assumed when data is fused despite multiple time sources.

- Virtualization and mixed-criticality applications: Mixed criticality applications such as periodic & event driven processing, scheduling priorities vs. load scheduling priorities, multiple security layers, safety-criticality levels, and redundancy requirements, must utilize virtualization consistently despite conflicting demands.

A fourth root cause areas is distributed and replicated state-based systems, such as discrete application logic, hand-shaking protocols, operational modes, and reconfiguration of operational systems. Assumptions made when validating such concurrent and stochastic state machines in a synchronous system setting without failure may not hold.

- Coordination of state machines: Communication of state vs. state transition events responds differently to protocol message loss.
- Event observations: Sampling of state to observe events is a common technique in periodically operating applications that assumes no event loss. It may have race conditions under concurrency, asynchronous clock, and faulty communication conditions. This may result in event observation loss and protocol lockup. Validation requires not just temporal ordering assumptions, but needs to address temporarily inconsistent time intervals.

The SAE AADL standard [4] is an architecture modeling language for embedded systems to capture the architecture of the computer platform, the architecture of the operational application, and the architecture of the physical system and their interactions. An AADL model may be used during a broad range of life-cycle activities, e.g. for documentation during preliminary specification, for schedulability or reliability analysis during design studies and during verification, for generation of system integration code during implementation.

AADL provides a set of concepts with well-defined semantics to represent a system as a component-based model. These allow us to capture relevant aspects of the embedded system in order to address these root cause areas analytically. AADL has

- threads, whose semantics are defined by hybrid automata in the standard, to represent concurrent tasks;
- processes to represent protected address spaces (space partitions);
- sampling and queued port connections with timing specifications, including deterministic sampling requirements to minimize jitter;
- virtual processors and virtual buses to represent partitions, hierarchical schedulers, protocols, and virtual channels;
- processors, buses, memory, and devices to represent hardware and physical system architectures;
- allocation bindings of software to hardware to represent deployment decisions;
- abstract specification of flows through components and end-to-end flows to support flow-related analyses;
- partial model specifications and their refinement to support evolution of models at multiple levels of granularity and architectural patterns;
- packages to organize models into manageable units that can be developed and analyzed independently by different team members and suppliers; and
- an extensible set of properties to annotate model elements with information relevant to different analyses with many properties already defined with the base standard and others standardized through annex documents.

In the next section, we will discuss how AADL is a driving force behind an architecture-centric approach to model-based analysis and construction of embedded systems.

ARCHITECTURE-CENTRIC SYSTEM VALIDATION

AADL supports the concept of an architecture model that is annotated with information relevant for analysis and validation of different operational quality dimensions. As such, it is the single source for analytical models of the same system. Their auto-generation ensures model consistency and single ”truth” of the analysis results (Figure 4). In addition, this single source approach facilitates automatic propagation of system architecture changes, such as the replacement of a processor component by a higher capacity one, into different analytical models, each addressing a separate quality aspect, e.g., the reconfigured processor is reflected in budget and scheduling analysis, as well as weight, and power consumption analysis.

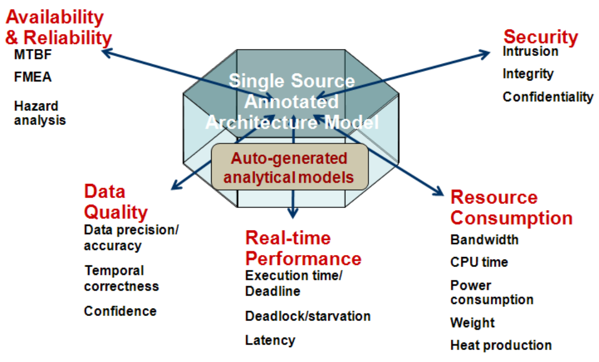

Figure 4: Single Source Annotated Architecture Model

AADL allows systems to be modeled at different levels of granularity and different levels of fidelity. Early in the process, we may have a model of the system in terms of major subsystems. We can associate resource budgets and resource capacities and perform budget analysis. In that context we can take into account, any deployment decisions of subsystems to hardware early in the system life cycle. On the same system model, we can perform initial end-to-end latency analysis based on the fact that different subsystems may be deployed as separate partitions in a partitioned architecture. In other words, the latency analysis not only takes into account processing and sampling latency from the control engineer perspective, but also considers latency and jitter contributions due to the software implementation and the computer hardware [20]. Early in the life cycle, we are able to detect, that migration to a partitioned architecture may increase latency of critical flows.

Once the subsystems are refined to the task level we can revisit the budget analysis, the latency analysis, and perform resource allocation and scheduling analysis. In other words, we can create initial models with limited effort and perform low fidelity quantitative analysis on a system. We then refine the model and the information we annotate the model with at incremental cost.

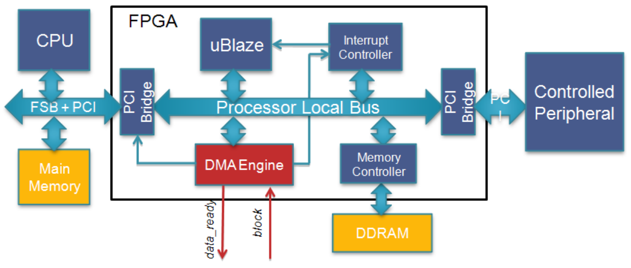

Figure 5: Detailed Hardware Architecture

The computer architecture is represented in AADL in terms of processors, memory, and switches (buses), a notation introduced as Processor Memory Switch (PMS) in [18], and supports relevant architectural detail to explore resource contention issues (Figure 5) [19]. When combined with allocation of tasks to processors, code and data to memory, and connections to buses, the resource demand in form of workload can be derived from the application properties.

Safety-criticality is supported by AADL in a number of ways. First, AADL is strongly typed. For example, A processor specification indicates that it requires access to a Peripheral Component Interconnect (PCI) bus and an Ethernet, then only a PCI bus can get connected to the one bus access feature. Second, the protected address space enforcement of processes and resource allocation enforcement of virtual processors ensure time and space partitioning. Third, a safety level property on system components, initially used for major subsystems and later attached to more detailed components, is used to ensure that components with high safety criticality are not controlled by or receive critical input from low criticality components. Fourth, AADL has a built-in fault-handling model for application threads and extends into an explicit representation of a health monitoring architecture, and fault management by reconfiguration, which is modeled through AADL modes. Fifth, AADL support fault modeling through the Error Model Annex standard [17], which allows us to introduce intrinsic faults, error state machines, and fault propagations across components – including stochastic properties such as probability of fault occurrence. These annotations to the architecture support hazard and fault impact analysis as well as reliability and availability analysis. Finally, the dynamic behavior of the architecture is represented by modes and further refined through the Behavior Annex standard [22]. This allows us to apply formal methods such as model checking to validate system behavior, as was done for the mode logic of a dual redundant flight guidance system (shown in Figure 6) [21].

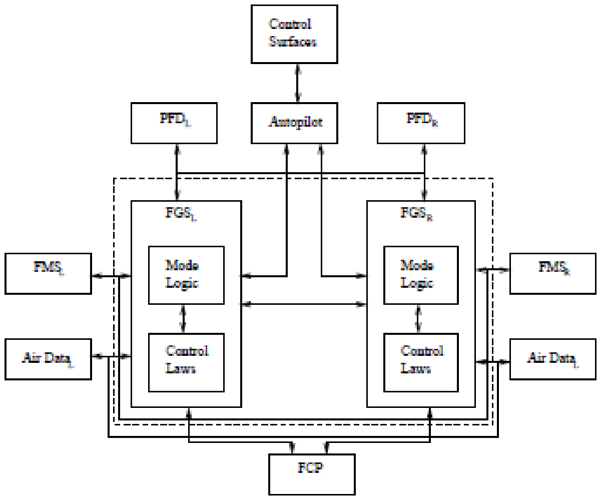

Figure 6: Dual Flight Guidance System

The architecture model can be further refined by associating detailed design models with individual components, such as Modelica models for physical components, VHDL models for hardware components, and Simulink or Scade models and Java source code for software components. This allows us to validate the detailed design against the architectural specification of components and their interfaces. Figure 7 illustrates this collaborative approach to engineering systems.

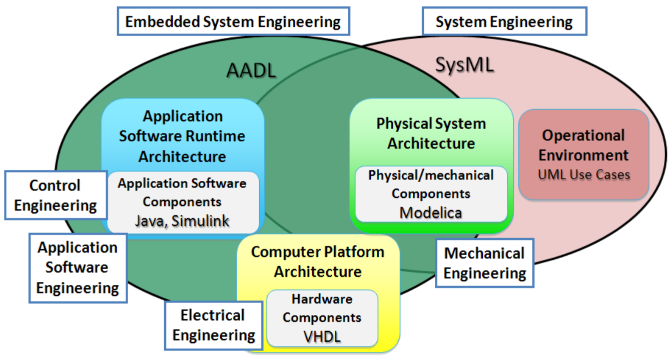
Embedded System Engineering
System Engineering
AADL
SysML
Application Software Runtime Architecture
Application Software Components
Java, Simulink
Physical System Architecture
Physical/mechanical Components
Modelica
Operational Environment
UML Use Cases
Control Engineering
Application Software Engineering
Computer Platform Architecture
Hardware Components
VHDL
Mechanical Engineering
Electrical Engineering

Figure 7: Co-Engineering of Systems

At the architecture level, it is desirable to combine AADL with the System Modeling Language (SysML) for collaborative engineering by system architects and embedded system architects. The focus of SysML is to represent the requirements, structure, behavior, and parametrics [23], in addressing multiple aspects of a system. The two standards working groups have started to collaborate in an alignment of the two notations.

CONCLUSION

In this paper, we have made the case that the embedded software system has become a major contributor to system-level faults in to today's systems. A range of system-level failures has been illustrated. The ability to represent embedded software systems in terms of concepts whose semantics are well defined, is key to understanding the root causes of such system-level faults and analyzing system models to validate its safety criticality requirements. We have shown that AADL is able to play this role and act as the single source architecture representation for analytical models.

The benefits of this model-based engineering approach with focus on the system architecture include:

- Reduced risk through analysis early and throughout the life cycle, understanding of system-wide impact of changes, and validation of assumption across the system architecture;
- Increased confidence through model validation to complement integration testing, validation of assumption made by models against the implementation, and analysis of evolving model into higher fidelity;
- Reduced cost through fewer system integration problems, and fewer validation steps through the use of a single source model repository and generation of analytical models as well as implementation code.

Recognizing the challenges of embedded software systems and the potential value of this architecture-centric approach, the Aerospace Vehicle Systems Institute (AVSI), a cooperative of aerospace companies, government organizations, and academic institutions, has launched an international, industry-wide program called System Architecture Virtual Integration (SAVI). Major players of the SAVI project include Boeing, Airbus, Lockheed Martin, British Aerospace Engineering (BAE) Systems, Rockwell Collins, General Electric (GE) Aviation, Federal Aviation Administration (FAA), Department of Defense (DoD), Software Engineering Institute (SEI), Honeywell, Goodrich, Hamilton Sundstrand, and National Aeronautics and Space Administration (NASA).

REFERENCES

1. NIST Planning report 02-3, "The Economic Impacts of Inadequate Infrastructure for Software Testing", May 2002.
2. VHSIC (Very High Speed Integrated Circuits) hardware description language en.wikipedia.org/wiki/VHDL.
3. Model checking hardware en.wikipedia.org/wiki/Model_checking.
4. SAE International. "Architecture Analysis & Design Language (AADL) Standard.", AS 5506A, November 2004. Revised Jan 2009.
5. "2006 Lathen maglev train accident", http://en.wikipedia.org/wiki/2006_Lathen_maglev_train_accident.
6. "Ariane 5 Flight 501.", en.wikipedia.org/wiki/Ariane_5_Flight_501.
7. "Air Canada Flight 143.", en.wikipedia.org/wiki/Gimli_Glider.
8. Allen Li. Testimony by Allen Li, Director, Acquisition and Sourcing Management, U.S. General Accounting Office, to the House Subcommittee on Tactical Air and Land Forces, Committee on Armed Services, April 2003. www.gao.gov/new.items/d03603t.pdf.
9. ITunes Crashes on Dual-core Processors. discussions.apple.com/thread.jspa?messageID=1235236&.
10. M. Jones, "What Really Happened on Mars", http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/.
11. Lui Sha, R. Rajkumar, and John P. Lehoczky (September 1990). "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". IEEE Transactions on Computers 39 (9): 1175–1185.
12. "History of the Internet.", www.thocp.net/reference/internet/internet2.htm.
13. Avionics Application Software Standard Interface. "ARINC 653 Standard Document." www.arinc.com.
14. Feiler P., "Upgrading Avionics Systems: A Case Study", DARPA EDCS Project Report, June 1998.
15. Cervin, A.; Årzén, K.-E.; & Henriksson, D. "Control Loop Timing Analysis Using TrueTime and Jitterbug," 1194–1199. Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design (CACSD). Munich, Germany, October 4–6, 2006.
16. "Qantas Flight 72.", en.wikipedia.org/wiki/Qantas_Flight_72.

17. SAE International, “Architecture Analysis & Design Language (AADL) Annex Volume 1: Error Model Annex"; SAE Document AS-5506/1, 2006 June.
18. C. Bell, A. Newell, “Computer Architectures: Readings and Examples”, McGraw-Hill, 1971.
19. Min-Young Nam, R. Pellizzoni, Lui Sha, “ ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture”, 14th IEEE International Conference on Engineering of Complex Computer Systems, 2009.
20. Peter H. Feiler, Jörgen Hansson, “Impact of Runtime Architectures on Control System Stability”, Proceedings of 4th International Congress on Embedded Real-Time Systems, Jan 2008.
21. de Niz, D. and Feiler, P. H., “Verification of Replication Architectures in AADL”. 4th IEEE International Workshop UML and AADL, *Proceedings 14th International International Conference on Engineering of Complex Computer Systems* (ICECCS 2009).
22. Fanca, R.B., et al. “The AADL Behavioral Annex - Experiments and Roadmap”. Proceedings of 12th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS07), UML&AADL Workshop, July 2007.
23. Systems Modeling Language (SysML). www.sysml.org.

CONTACT

Peter H. Feiler, Software Engineering Institute, Carnegie Mellon University. phf@sei.cmu.edu. Technical Lead of SAE AADL standard.

DEFINITIONS, ACRONYMS, ABBREVIATIONS

AADL: Architecture Analysis & Design Language

ADIRU: Air Data Inertial Reference Units.

AVSI: Aerospace Vehicle Systems Institute.

BAE: British Aerospace Engineering.

COP: Common Operational Picture.

DMA: Direct Memory Access.

DoD: Department of Defense.

DPCP: Distributed PCP.

FAA: Federal Aviation Administration.

GAO: General Accounting Office.

GE: General Electric.

IMA: Integrated Modular Avionics.

MMC: Modular Mission Computer.

NASA: National Aeronautics and Space Administration.

NIST: National Institute of Standards and Technology.

PCI: Peripheral Component Interconnect.

PCP: Priority Ceiling Protocol.

PMS: Processor Memory Switch.

RMS: Rate-Monotonic Scheduler.

RTOS: Real-Time Operating System.

SAVI: System Architecture Virtual Integration.

SEI: Software Engineering Institute.

SysML: System Modeling Language.

UML: Unified Modeling Language.

VHDL: VHSIC (Very High Speed Integrated Circuits) Hardware Description Language.