Integrate Mixed-Criticality Components with Formalized Architectural Patterns[1]

Lui Sha

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

1. Introduction

How to integrate mixed criticality components is a major challenge in CPS transportations. Our inability to verify highly complex components is at the heart of this challenge. Unfortunately, in many practical applications, we cannot avoid the use of components whose correctness, including safety and liveness, is either impossible or impractical to verify.

Example 1: After major surgery, a patient is allowed to operate an infusion pump with potentially lethal pain killers (patient controlled analgesia (PCA)). When pain is severe, the patient can push a button to get more pain-relieving medication, e.g., morphine sulfate. This is an example of a safety critical device operated by an impossible to verify component, the patient. Nevertheless, the PCA system as a whole needs to be certifiably safe in spite of a patient's unsafe actions.

Example 2: As illustrated in Figure 1, in a flight control system, the autopilot is certified to DO 178B Level A, the highest safety critical level, while the flight guidance system, because its complexity, can only be certified to Level C. Nevertheless, the Level C guidance system issues commands to steer the Level A autopilot. This is an example of safely using a component whose correctness is impractical to verify under current technologies. Nevertheless, the overall flight control has to be certified to Level A.

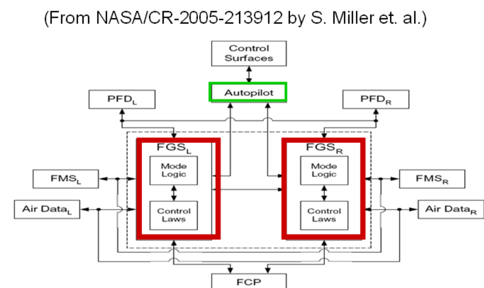

Figure 1: Flight guidance System (FGS) - Autopilot interaction

The current approach to solve these problems is case by case. What is needed is to develop formally verifiable architecture patterns, each of which can solve a family of similar problems. We propose a novel paradigm, based on the idea of "*using simplicity to control complexity*"[1], that will allow us to safety use unsafe components. To illustrate this idea, let us consider the problem of sorting. In sorting, the critical property is to sort items correctly. The desirable property is to sort them fast. Suppose that we could formally verify a Bubble Sort program but were unable to verify a ComplexFastSort program. Can we safely use the unverified ComplexFastSort? Yes, we can.

As illustrated in Figure 2, to guard against all possible faults of ComplexFastSort, we put these two programs in two virtual machines. In addition, we develop a verified object called "permute" that will: (i) allow ComplexFastSort to perform all the list operations to rearrange the order of the input item in the input list, but *not* to modify, add or delete any list item; and (ii) check in linear time that the output of ComplexFastSort is indeed sorted. Finally, we set a timer based on the promised speed of ComplexFastSort if it is supposed to be faster than the BubbleSort. If ComplexFastSort does finish in time and we check that the answer is correct, then the result is given as output; if it does not finish in time or does so but with an incorrect answer, then BubbleSort sorts the data items.

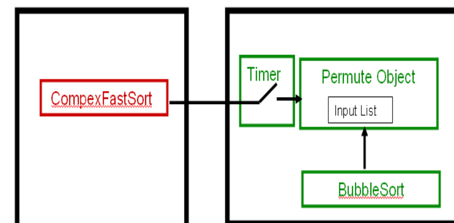

Figure 2: Always Correct Sorting System

[1] This position paper is a abbreviated version of "Design of Complex Cyber Physical Systems with Formalized Architectural Patterns" by Lui Sha and Jose Meseguer, to appear in the proceedings of 2008 Interlink Workshop, the European Consortium for Informatics and Mathematics. http://www.ercim.org/content/blogcategory/29/97/

It is important to note that this sorting system is provably correct for all possible new sorting components, including the use of a human to do the sorting. Furthermore, there is a lower bound on performance and this lower bound can be improved by replacing BubbleSort by a faster and formally verified sorting program. The moral of this story is that we can safely exploit the features and performance of complex components even if it may have unsafe behaviors, as long as we can guarantee the critical properties by simple software and an appropriate architecture pattern. In this way we can leverage the power of formal methods to provide high assurance in the system development process. We call this architecture principle "using simplicity to control complexity."

Checking the correctness of an output before using it, such as in the sorting example, belongs to a fault tolerant approach known as recovery block [19]. However, in CPS applications, it may *not* be possible to determine if a command from a complex controller is correct (meeting the specifications). Fortunately, it is safe to execute a control command whose correctness cannot be determined, provided that we can determine the resulting state is still within the stability margin[1]. The simplex architecture allows us to safely exploit complex high performance control subsystems that may have residual errors by using a simple high assurance subsystem and by monitoring the resulting stability margin if a command were executed (Figure 3) [1]. That is, if a command might lead to instability, we always reject it; otherwise we give it the benefit of doubt.

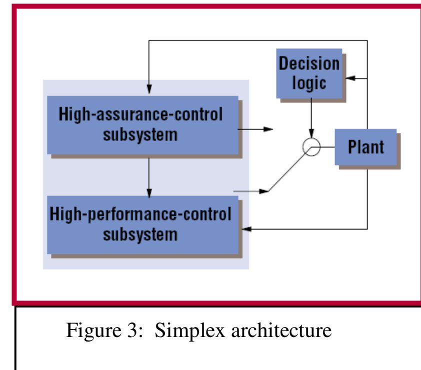

Figure 3: Simplex architecture

A noteworthy real world example of "using simplicity to control complexity" in practice is the flight control system of Boeing 777 [18]. It uses triple-triple redundancy for hardware reliability. At the software application level, it uses two controllers. The sophisticated control software specifically developed for Boeing 777 is the normal controller. The secondary controller is based on the control laws originally developed for Boeing 747. The normal controller is much more complex and is able to deliver optimized flight control over a wide range of conditions. On the other hand, control laws developed for Boeing 747 have been used for over 25 years. It is a mature old technology – simple, reliable and well understood. From our perspective, we will call it a simple component, since it has low residual complexity[2]. To exploit the advantage of advanced control technologies and to ensure a very high degree of reliability, Boeing 777 under the normal controller should fly within the stability envelope of its secondary controller. This is a fine example of using simplicity to control complexity.

In summary, complex and unverifiable components, e.g., human operators and highly complex software components, are unavoidable. Fortunately, we can ensure critical properties and lower bounds on performance using: (1) formally verified complexity control architecture patterns, and (2) formally verified simple components for essential services. That is, under a given fault model we need to verify the following properties:

- Protection: The architecture software and the simple component cannot be corrupted by faults from the unverified complex software components.
- Timeliness: The simple and verified components must be executed within timing constraints.
- Fault tolerance: in spite of all the faults under the fault model, the simple, verified component will function correctly.

Since architecture patterns often need to be adapted for new application requirements, we need to not only verify a collection of commonly used architecture patterns, but also provide computer aided verification for the adaptation of architecture patterns. Furthermore, since in software practice model-based approaches are the most common way of capturing architectural designs and architectural patterns, it is important to provide formal verification support for architecture patterns expressed in software modeling languages.

To make all this possible Lui Sha, Jose Meseguer, Marco Caccamo and their students collaborate with Artur Boronat at the University of Leicester; Peter Olveczky at the University of Oslo; Steve Miller and Darren Cofer of Rockwell Collins; Ben Watson, Jonathan Preston and Russell Kegley of Lockheed Martin; Dr. Julian Goldman of Massachusetts General Hospital and MDPnP.org; and Peter Feiler, Jorgen Hansson and Dionisio de Niz of Software Engineering Institute on several mutually-reinforcing tasks:

- Complexity control architectures and design rules for avionics and medical systems.
- Formalized SAE AADL [3] subset to specify these architectures.

[2] Logical complexity of a software system is can be measured by the number of states that we need to check. A program could have high logical complexity initially. However, if it has been formally verified and can be used as is, then its residual logical complexity is zero

- Use of MOMENT2 [6][8] to automatically transform AADL models into algebraic expressions in Maude for formal analysis purposes and for further transformation into Real-Time Maude [9] specifications.
- Formal semantics of AADL in Real-Time Maude, and automatic transformation of AADL models into Real-Time Maude specifications based on such semantics, to provide both symbolic simulation and formal verification by model checking for AADL models.
- Embedding the high assurance control subsystem in FPGA by directly generating VDHL code from verified AADL specifications. This approach makes the high assurance subsystem immune to OS faults.

Acknowledgement: Works described here are supported in part by NSF, ONR, Rockwell Collins, Lockheed Martins, and Software Engineering Institute.

References

[1] L. Sha, Using Simplicity to Control Complexity, IEEE Software, July/August, 2001. https://agora.cs.uiuc.edu/download/attachments/10581/IEEESoftware.pdf?version=1
[2] *Fault Tolerance*, John Wiley & Sons, 1995. Editor Lyu, M. R.
[3] AADL: http://www.sei.cmu.edu/products/courses/p52.html
[4] M. Clavel, F. Duran, S.Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott,
[5] All About Maude – A High-Performance Logical Framework, Springer LNCS 4350, 2007.
[6] MOMENT2 : http://www.cs.le.ac.uk/people/aboronat/tools/moment2-gt/
[7] G. Rosu and K. Havelund, Rewriting-Based Techniqes for Runtime Verification, Automated Software Engineering, 12, 151-197, 2005.
[8] Boronat and J. Meseguer: An Algebraic Semantics for MOF. In Proc. FASE 2008, 377-391, Springer LNCS 4961, 2008.
[9] P. C. Ölveczky and J. Meseguer: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2): 161-196 (2007).
[10] G. Behrmann, A. David, and K.G. Larsen, A Tutorial on UPPAAL, in Proc. SFM-RT 2004, 200-236, Springer LNCS 3185, 2004.
[11] T. A. Henzinger, P.-H. Ho, H. Wong-Toi: HYTECH: A Model Checker for Hybrid Systems, Softw. Tools Technol. Trans., 1, 110-122, 1997.
[12] S. Yovine, Kronos, A Verification Tool for Real-Time Systems, Softw. Tools Technol. Trans., 1, 123-133, 1997.
[13] J. Misra, A Discipline of Multiprogramming, Springer, 2001.
[14] M. Viswanathan and R. Viswanathan: Foundations for Circular Compositional Reasoning, in Proc. ICALP 2001, 835-847, Springer LNCS 2076, 2001.
[15] I. Poernomo, The meta-object facility typed. In Proc. SAC, 1845–1849, ACM, 2006.
[16] J. R. Romero, J.E. Rivera, F. Duran, A. Vallecillo, Formal and Tool Support for Model Driven Engineering with Maude. Journal of Object Technology 6(9), 2007.
[17] M. Lyu, Software Fault Tolerance, http://www.cse.cuhk.edu.hk/~lyu/book/sft/index.html
[18] Y.C. Yeh, "Dependability of the 777 Primary Flight Control System," Proc. Dependable Computing for Critical Applications, IEEE CS Press, Los Alamitos, Calif., 1995
[19] A.M. Tyrrell, A. M. Tyrrell Recovery Blocks and Algorithm-Based Fault Tolerance, Proceedings of the 22nd EUROMICRO Conference