Architecture and Design Modeling and Simulation Testbeds for NASA: A Progress Report

Daniel Cooke
Texas Tech University, Lubbock, TX 79409

Michael Evangelist
Carnegie Mellon University, Moffett Field, CA 94035

Dewayne E. Perry
University of Texas, Austin, TX 78712

I. Introduction

The software-intensive exploration systems of the future will be highly complex, and their operation will be exceptionally visible to the nation. In addition to providing complex functionality, they must tolerate the subtle faults of asynchronous systems running in a hostile environment and be affordable, reliable, flexible, robust, and gracefully upgradeable, among other things. In addition to all the usual problems of complex software, exploration systems will have to accommodate, for example: (in some missions) significant communication latencies, requiring more autonomy; diminished communication windows and bandwidth, requiring improved data reduction and compression; the need to reduce the number of in-flight updates, because of the risk at each update; and the need to reduce the systems-administration burden on astronauts, because their time is a scarce resource.

Achieving all these characteristics using the current generation of techniques and engineering-support technologies for building software-intensive systems would be too expensive and time-consuming, and the cost and time required would severely limit system capability. We must find ways to engineer dependable and resilient exploration architectures that significantly reduce risk.

Our research group is developing ways to engineer “product lines” of sophisticated software/hardware testbeds to support: (1) the iterative improvement of new architectural techniques for building software-intensive exploration systems; (2) evaluations of the techniques and support technologies to help understand the risk, cost, scope of applicability, and benefit of using them; and (3) for mission operators, a knowledge-base of testbed-generated information and support technology to isolate the cause of bugs that emerge during flight and remove them quickly and precisely.

This work builds on results from the NASA-sponsored High Dependability Computing Program (HDCP), which has helped mature the concept of testbeds for software-intensive systems, demonstrating both their use and their value. [1, 2, 3, 4] Our testbeds combine hardware and software, especially embedded control systems, to represent relevant mission functions. To be a testbed and not merely a capability demonstration, the combination must be instrumented for gathering operational data to assess reliability,

effectiveness, and other important characteristics, which will help when deciding among candidate solutions to a mission problem.

Two HDCP testbeds are especially relevant to this work: the Dependable Real-Time Software testbed, which controls a rover that is helping us investigate the effectiveness of new approaches to programming real-time-control systems, and the Dependable Automated Air-Traffic Management testbed, which helps evaluate and improve software for automating critical flight activities. These projects led to the creation of our Testbed Engineering Platform (TEP), allowing us to rapidly build just-in-time testbeds for a large variety of mission applications. TEP substantially reduces cost and time in creating testbeds.

TEP and the testbeds it helps create are part of an experimental methodology we are developing to assess and improve new software-engineering techniques and technologies. This paper is a progress report on our approach.

II. Architecture

Software architecture (SA) represents the structure of a software system and defines the essential components in a system, their critical constraints and interactions. SA is the system blueprint and the engineering basis for design, coding, testing, integration, estimation, and planning. Without an explicit and carefully defined architecture, risk assessment and mitigation is infeasible.

The importance of SA here is two-fold: first, it is of critical importance for the test-bed itself; second, it is important for NASA and the software systems to be used in coming missions. It is of particular importance for the testbed technology, because it provides a means—using a product-line architecture—of quickly responding to the needs of new testbeds. Engineering the testbeds into a product-line framework will give NASA an even more agile and cost-effective capability for evaluating new technologies, designs, and engineering processes for exploration missions.

Within NASA itself there is a recognition that software architecture is a critical ingredient in its software-intensive systems. In the Fall of 2003, the Texas Tech Workshop on NASA Control Architectures determined that the architectures discussed have not been formally studied—an important precursor to understanding and mitigating risk. Since that time, effort has been put into organizing the elements of control systems better. This improved organization together with the use of the industrial-proven techniques mentioned below allows for a more-refined deployment of capabilities of varying complexity, making it easier to develop, verify, and maintain control systems.

Four software-architectural themes are important for mission-software developments:

1. software architecture as an industry best practice;

2. software architecture as a development-coordination principle;

3. product-line architecture as an engineering mechanism for managing diverse but related software-intensive systems; and

4. model-based, self-managing software architectures as a promising mechanism for mission support.

Just as software architecture plays a critical role in the development and evolution of all successful complex systems, software architects play a fundamentally important role in the development and evolution of such systems. They structure and integrate the system in both the technical and the political sense.

Architecture-driven development is widely considered to be a best practice in industry. AT&T/Lucent, for example, defined the requirements that had to be satisfied by various development processes to be considered in the best-practice class of processes. The most critical of those process requirements was the need for an architecture-driven development and evolution process. Indeed, studies have shown [5] that architecture-driven development and evolution results in faster development and higher quality.

Furthermore, empirical studies have shown [5] that a software architecture is a critical element in coordinating distributed development and evolution of software systems. The architectural interfaces, in particular, become the critical coordinating element in developing software systems by geographically separated teams, organizations, and companies. A prime advantage of such an organizing structure is that the different participants can use independent operating environments—they do not need to have the same processes and tools. Nor do they need to have the same project-management and -quality structures. Architecture-based coordination can accommodate different cultures and organizational structures. (See, for example, [5].)

Real-world software-engineering projects have demonstrated repeatedly that the product-line architecture approach can be extremely useful for building and evolving diverse systems. For example, one industrial project [6] used the product-line concept to create a reference architecture that represented an extremely diverse set of architectures, ranging from a simple centralized system to a complex, distributed multi-processor system. By focusing the reference architecture on critical issues, collecting the common components into an asset base for the individual architectures, and explicitly managing individual-product diversity, product-line architecture could provide a cost-effective mechanism for related NASA software-intensive systems.

In the workshop on Software Engineering Technology, held at the Lunar and Planetary Institute in April 2004, the development of model-based approaches that automatically discover correct algorithms and solutions from very high-level specifications was recommended as necessary for the anticipated exploration missions, which are less pre-scripted than previous missions. They will, therefore, require the rapid development of capabilities between—and possibly even during—missions. Not only will model-based approaches reduce the risk associated with software, they will allow for new approaches to risk management focused on the capability level rather than the software level.

These model-based and product-line approaches then provide the foundation for self-managing architecture, a highly promising (if not yet fully proven) concept that may become fundamentally important. Exploration missions, which will be long-lived and will frequently have significant communication latency, need on-board software-intensive systems that can dynamically manage repair and reconfiguration without the presence of a highly knowledgeable software staff. The flight team can concentrate on other critical mission issues while the self-managing architectures concentrate on keeping the systems fully functional and available.

III. Example Architecture Testbed

In a companion paper [7] we present a language architecture for the rapid development of flight- and mission-control software. The architecture grew out of the observation that although the controls architectures, 3-T, CLARAty, and MDS, have worked well in the past, a formal understanding of the architectures had not been developed. The Texas Tech architecture also grew out of the observation that future systems will require continual modification in order to respond to unforeseen emergencies and exploration opportunities. The components of the language architecture are shown in Figure 1.

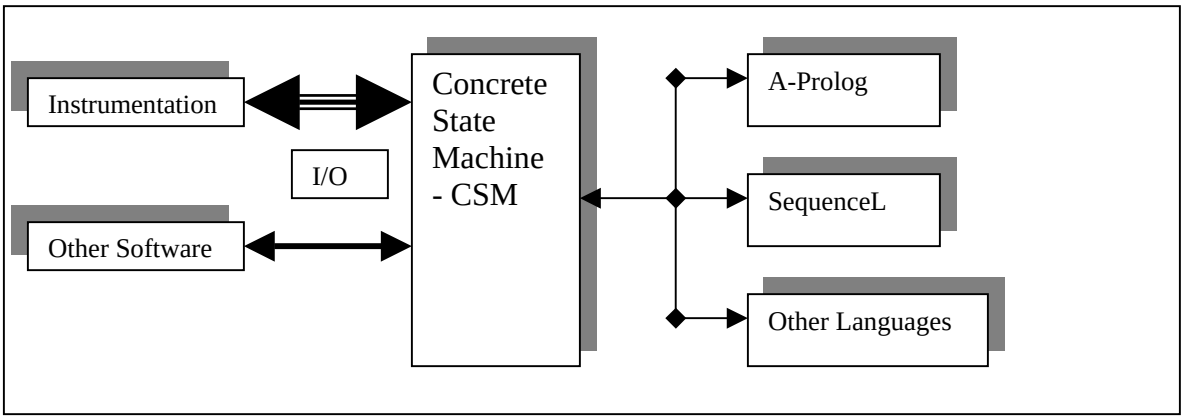

Figure 1. Language Architecture and Its Planned Environment.

The CSM, SequenceL, and A-Prolog [8, 9, 10, 11, 12] languages have been developed by the Declarative Language Group at Texas Tech University. The A-Prolog effort has resulted in a high-level language in which onboard systems can be modeled. For example, working with United Space Alliance, the group has developed the USA-Advisor. This system includes a model of the Reaction Control System and is capable of finding in a matter of seconds, provably correct work-arounds in the presence of multiple subsystem failures. Systems such as these will need to be deployed onboard future missions to provide some of the mission-control capabilities when time delays will make it difficult to communicate with the ground when certain emergencies occur.

The SequenceL language [13, 14, 15, 16] is a Turing-complete executable requirements language. We are currently using it to develop the Shuttle Abort Flight Management

System requirements (SAFM). Although SAFM, is already in final revisions, the SequenceL effort will provide a testbed to show that in the future it may be unnecessary to develop costly and time-consuming programming-level implementations of prototype Guidance, Navigation, and Control systems. Instead, through the use of the concise SequenceL language, simple requirements can be formulated and then executed in the language architecture. The SequenceL language translator automatically discovers much of the control-structure content of the algorithms satisfying the requirements.

Many prototype validation tasks can be accomplished using SequenceL in isolation. However, to move the system requirements into a testbed environment requires the CSM language, which facilitates I/O for A-Prolog and SequenceL. This language is the only part of the Texas Tech language suite that allows for assignment to variables. It is based upon Gurevich's [17] abstract state machines. The language provides only the concurrent, conditional control of input-output. CSM is extremely small, invoking SequenceL for computations and A-Prolog for deliberation. Since it is small and based upon an excellent semantic foundation, the major difficulties involved in verifying traditional procedural or Object-Oriented codes are avoided. SequenceL produces the lion's share of the resulting programs, but does so in such a way as to prevent the programmer from having to write much of the error-prone nested control structures. Instead SequenceL discovers the correct control structures automatically.

We plan to use the Testbed Engineering Platform to create testbeds to facilitate the evolution of the SequenceL/CSM requirements—providing high-fidelity simulations prior to committing the developed systems to the more costly simulators at Johnson Space Center. In other words, the testbeds will provide Texas Tech researchers with the ability to evaluate their results at the university, prior to more extensive testing in the Shuttle Engineering Simulator at JSC. More generally, our architecture testbeds will evaluate and improve both the SequenceL language concepts and compilers for generating efficient, maintainable implementations.

IV. Future Plans

In the following, we describe testbed projects currently under development or discussion. Depending on funding, some subset of these will emerge as full-blown testbeds over the next several years:

1. Identifying weak points in software-intensive exploration systems for thorough testing. Testbed evaluation of new architectural principles, algorithms, and technology employed on a software project will help make Verification & Validation more productive by identifying the weak points that need to be investigated in implementation testing. We've explored all of these issues on previous testbeds but not yet for the purpose of localizing potential problems in a final implementation. This project should lead to a new problem-driven approach to V&V.

2. Debugging of very complex asynchronous systems. Debugging of asynchrony is notoriously difficult, because you cannot obtain an instantaneous snapshot of the global state without "freezing" the system and eliminating its asynchrony. Testbeds that abstract

away the clutter will help pinpoint likely problem areas in the implemented system and make debugging simpler (though certainly not simple).

3. Software reuse and integration of systems with other complex systems. We will investigate and evaluate proposed integration styles at decreasing levels of abstraction on a systematic succession of testbeds.

4. Fault tolerance. We plan to use testbeds to adapt and mature novel approaches to fault tolerance such as self-healing and self-stabilizing systems.

5. Tolerating communication latencies. The existence of significant communication latencies implies at least two interesting problems. Exploration systems will require: (1) various kinds of autonomy; and (2) out-of-phase software components to work together efficiently. We will evaluate autonomy claims using flight and robotics testbeds. We've explored time-criticality in other testbeds and will apply lessons-learned on testbeds when latencies are in minutes rather than seconds.

6. Minimizing the number of software upgrades, because of the danger of mid-flight changes. You minimize the need for upgrades through a careful engineering process. We've already shown that testbeds provide the right controlled environment for exploring new engineering techniques (for example, our Dependable Real-Time Software testbed).

7. Making software upgrades fail-safe. This is a primary application of testbeds, and we plan to use testbeds to evaluate formal approaches to failure-proofing.

8. On-board ability to repair software for working around in-flight problems. "Testbedding" to identify and solve in-flight problems in real time by combining data from testbed evaluations and V&V testing is an exciting idea. It could have great value in long-lived exploration missions.

9. Hardware could be years out-of-date by the time a mission reaches Mars. We will use testbeds to evaluate software renderings of new functionality that would otherwise be implemented in hardware. Realistic testbeds will help us understand the special stresses on the implementations.

10. Minimizing communication. We need new approaches to communication minimization, given the massive amount of science data generated and the limited bandwidth. We plan to put together testbeds for evaluating new data-compression and communication-protocol ideas as they emerge.

11. Maturating and evaluating new approaches to software architecture. Because the exploration missions will last into the distant future, they must use leading-edge (but not bleeding-edge) architectural styles. Therefore, they need rigorous testbed evaluations of the applicability and efficacy of new architectures. For example: (a) Service-oriented architectures of the type we are helping build for Integrated System Health Management. (b) Advanced fault-tolerant architectures. (c) The product-lines architectural approach,

now widely used in industry, which enables the generation of individual "products" from a generic and common architectural base.

12. In particular, developing architectures for long-lived mission software. Exploration software will be much longer-lived than for other manned missions. This is both an engineering-process and a fault-tolerance issue. We are developing testbeds, such as the SequenceL testbeds described previously, to evaluate and certify the new ideas.

13. Integrating human and technology activities that occur at great distances or in hostile environments. Well-instrumented testbeds are also the right framework for evaluating computer-human interfaces, because of the ease of data collection from experiments.

14. Prototyping, from requirements to implementations. We use testbeds to prototype at every stage of the development to answer such questions as: Is this requirement achievable with the resources envisioned? Does this architectural style meet the specified reliability needs (again, cost-effectively)? Is this programming language appropriate under precisely specified constraints? Do these algorithms perform as required? Our Dependable Real-Time Software and Dependable Automated Air-Traffic Management testbeds are good examples that we plan to elaborate.

15. Determining whether a technology meets human-rating requirements. To evaluate whether a space software system has the right design features for protecting astronauts and recovering from emergencies, we plan to create evaluation testbeds and measure the results of critical scenarios, both emergency and routine. Again, because of the exceptional level of instrumentation and the ease of data collection, our testbeds are well suited for determining and *documenting* whether a software technology: (a) tolerates two failures; (b) properly monitors critical functions and informs the crew of problems; (c) detects, isolates, and recovers from faults; and (d) has the appropriate capability for autonomous operation of critical functions.

References

[1] Gregory Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Programming with Non-Heap Memory in the Real Time Specification for Java. *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, California, 361–369.

[2] Daniel Dvorak, Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Project Golden Gate: Towards Real-Time Java in Space Missions. *Proc. 7th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC 2004)*, Vienna, Austria, 2004, 15–22.

[3] V. Kotov and V. Mehrotra. Redwood: Dependability Testbed Instrumentation Platform. *Proc. International Conference on Dependable System and Networks*, June 28–July 1, Florence, Italy, 2004, 20–24.

[4] V. Kotov and V. Mehrotra. A Collaborative Testbed Engineering Platform. *Information Journal*, Vol. 8, No. 5 (September 2005).

[5] R. E. Grinter, J. D. Herbsleb, and D. E. Perry. The Geography of Coordination: Dealing with Distance in R&D Work. *Proc. GROUP '99*, Phoenix, AZ, November 14–17, 1999; online at: http://www.ece.utexas.edu/~perry/work/papers/DP-99-sgw.pdf.

[6] Dewayne E. Perry. A Product Line Architecture for a Network Product, ARES III: Software Architectures for Product Families 2000, Los Palmos, Gran Canaria, Spain, March 2000. Springer-Verlag, LNCS 1951. p39-52; online at http://www.ece.utexas.edu/~perry/work/papers/DP-00-ares3.pdf

[7] Daniel Cooke, Michael Gelfond, Howard Hu, and J. Nelson Rushont. Application of Model-based Technology Systems for Autonomous Systems. *Proc. Infotech@Aerospace* (American Institute of Aeronautics and Astronautics), 2005.

[8] Marcello Balduccini and Michael Gelfond. Model-Based Reasoning for Complex Flight Systems. *Proc. Infotech@Aerospace* (Amer. Inst. of Aeronautics and Astronautics), 2005.

[9] Marcello Balduccini. USA-Smart: Improving the Quality of Plans in Answer Set Planning. *Proc. PADL'04*, Lecture Notes in Artificial Intelligence (LNCS), June 2004.

[10] Marcello Balduccini and Veena S. Mellarkod. CR-Prolog with Ordered Disjunction. *Proc. International Workshop on Non-Monotonic Reasoning,* NMR2004, June 2004.

[11] Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4-5):425-461, July 2003.

[12] Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. *Proc. AAAI Spring 2003 Symposium*, 2003, 9–18.

[13] Daniel E. Cooke. An Introduction to SEQUENCEL: A Language to Experiment with Nonscalar Constructs. *Software Practice and Experience,* Vol. 26, Issue 11 (November, 1996), 1205–1246.

[14] Daniel E. Cooke and Per Andersen. Automatic Parallel Control Structures in SequenceL. *Software Practice and Experience*, Vol. 30, Issue 14 (November 2000), 1541-1570.

[15] Daniel E. Cooke and J. Nelson Rushton. Iterative and Parallel Algorithm Design from High Level Language Traces. *Lecture Notes in Computer Science,* Vol. 3516, Apr. 2005, 891–894.

[16] Daniel E. Cooke and J. Nelson Rushton. Normalize, Transpose, and Distribute: A Basis for the Decomposition and Parallel Evaluation of Nonscalars. In revision for *ACM Transaction on Programming Languages and Systems.* http://www.cs.ttu.edu/~dcooke/sequencelrevjune22.pdf.

[17] Andreas Blass and Yuri Gurevich. The Linear-Time Hierarchy Theorems for Abstract State Machines and RAMs. *J. UCS* 3(4): (1997), 247-278.