Systems of Systems and Coordinated Atomic Actions

Robert Schaefer
BAE SYSTEMS
robert.p.schaefer@baesystems.com

ABSTRACT

System of systems (SoS) is an emerging field in the design and development of complex systems that are built from large scale component systems. A SoS has the following attributes: operational and managerial independence of components, a geographic extent that limits control mechanisms to information exchange, an evolutionary nature, and emergent behavior. The subsystems that comprise the SoS often are built by different organizations with conflicting goals, designed under different assumptions and built to different quality standards. These factors impact fault detection, fault isolation, and fault tolerance and can result in systems that cannot easily be debugged, integrated, or maintained. When fault detection and fault tolerance are deficient, the system may behave in a fragile or brittle manner, randomly and repeatedly crashing. Crashes prevent automated diagnosis algorithms from being executed and can prevent manual root cause analysis by erasing system state. Fragility during system integration can prevent achieving schedule milestones and deadlines. Deficient fault detection and fault isolation also impacts end users and system maintainers.
(Think <insert name of infamous project here>).

From the system architect's point of view, designing a system that can detect all possible fault conditions across all components can be an extremely difficult, if not impossible challenge. Can any system be trusted to diagnose or repair itself when it has been corrupted by faults? How do you prevent local faults from growing into global failures? The end users may have unreasonable expectations about how the system should behave when components within the SoS behave abnormally or fail. They may expect better behavior than the typical PC. The system maintainers may expect a coherent systems view of failures to isolate faulted components and to provide an orderly and safe shutdown or recovery.
(Think power grid blackouts, Telecomm failures, etc.)

The most beneficial way to achieve fault tolerance is to design in fault detection and fault reporting such that defined boundaries such as subsystems serve as natural firewalls for fault containment. Although partitioning the system into subsystems for fault containment is well known and practiced, the end result as experienced at the time of system integration is rarely a success. COTS middleware, intended to aid distributed design often becomes in effect a step backwards by providing fertile ground for faults and failures that breach fault containment boundaries.
(Think <insert name of OS or middleware vendor here>)

What can be done to improve this situation? This paper addresses the system architectural partitioning concept of the Coordinated Atomic Actions (CAA). CAA promotes a different manner of organizing software architecture that improves fault containment across potentially faulty components. CAA was first invented by members of Brian Randell's research group at the University of Newcastle at Tyne in the mid 1990's. CAA promotes the concept of the "transaction" which has been traditionally identified with database applications. When you access your bank account via ATM, you are exercising database transactions within your bank's financial SoS. CAA applies transactions to cooperating concurrent distributed processes, which are the basis for most large complex computing systems.

INTRODUCTION

The grand motivation for this paper is simply to try to get my hands around the concept of balancing complexity, software architecture, distributed processes, hardware and software faults, budget and schedule, quality. Why settle for the easy problems? The specific itch for this paper comes from having participated in the systems integration of four large difficult software intensive projects and then wondering if we could do better by thinking differently, not just by doing more of the same. In tying this together, I would like to address the relationship between Systems of Systems, systems integration, and fault tolerant design as a way of introduction to the concept of Coordinated Atomic Actions (CAA). CAA appears to map well to these challenges, and may even solve some of them. The references at the end of this paper should provide a starting point for further investigation. I will be up front and tell you that as a working software engineer I am not in the practice of writing papers for publication, and am quite astonished at having this, my first, published. Some caveats: I do use the spell checker, but actively ignore the suggestions from the dancing paperclip, and I caution my readers that this paper contains my opinions only, and that in no manner, shape or form, should these opinions ever be considered those of my employer. In any case, if you do follow along, I hope to make the information presented useful and perhaps even provide an "A-ha!" moment to those of you who support systems integration for large software intensive projects.

SYSTEMS OF SYSTEMS

For those who are not yet aware of or involved in developing Systems of Systems, these kinds of systems have the following attributes:

1. Independence of components in design, management, and operation
2. A geographic extent that limits control to messages over a network
3. Evolutionary nature
4. Emergent behavior

These attributes will be addressed not from the point of view of the systems architect, but from the point of view of the systems integrator. The systems architect works with the SoS as a concept, but the systems integrator works with the actual system as implemented. That is, the architect works with his (or her) head in the clouds, but the integrator works in the mud of the real world. Now, concerning the first attribute, independence of components in design may lead to independence of components in development. For the systems integrators, this can mean the responsibility to correct problems that weren't their own doing, which in turn, will entail negotiation of changes across multiple corporations' design teams. Negotiation takes time; something there is not much of during systems integration. The implication of the second attribute is that the systems integrator may need to walk outside or make a phone call to another human in order to reset a system or subsystem. Note that after delivery to the customer, walking outside or making a phone call may no longer be possible, especially when a component is under the ocean or in outer space. Reliability becomes more noticed with components of this sort. The third attribute has the implication, that for systems that take years to develop, the rationales that may have existed at the start of design may no longer be true at the start of systems integration. Some of the truths contained in the early documents can become suspect. There may be requirements and design whose utility, in metaphor, has become identical to that of the human appendix. It's there, but no one knows what it really does, and it shouldn't be removed until it becomes project-life-threatening. The fourth attribute, emergent behavior, is behavior that is not explicitly defined in any specification, and as a result, leaves the systems integrator with the oft asked question, "Is it a bug, or a feature?" New requirements and design may be needed to cope with unwanted emergent properties; again, something for there is not much time.

SYSTEMS INTEGRATION

Systems, and systems of systems need to be integrated, and systems integration is, perhaps, the most challenging part of project development. This claim can be made for the principal reason that there is no time or budget to correct large problems that have slipped through the previous phases of the development process. The basic assumption at the start of systems integration is that there are no remaining large problems, but whether this assumption is correct or not, there will be some number of faults remaining. Each fault may prevent subsystems from being integrated. Each fault needs to be isolated and analyzed in a prioritized manner so that the most critical faults will be addressed first. How many small faults equal one large problem? And how did all those faults get into the system in the first place? How can individual subsystems be made more reliable prior to systems integration?

The short answer is that faults can be prevented to some degree through both proper construction and fault tolerant design. Fault prevention by construction is done by adhering to the engineering development processes, but human beings being the fallible creatures that they are, faults will always remain to be discovered at integration. Over the past decade, effort in improving the software development process has been getting most of the attention and beneficially has resulted in tremendous bang for the buck. Fault tolerance by design has been seen as primarily a hardware issue, which is very well understood, if sometimes poorly implemented. The hardware reliability of individual components can be predicted using models and calculations based on the expected environment and the failure rates of the individual components. Fault tolerance is provided by proper choice of parts and by design redundancy. In contrast, there is no easy way to predict software reliability by apriori model or calculation. One can make an initial prediction of so many faults per thousand lines of code based on statistics from previous projects. This prediction will suffer due to the ever unique project context and the variability of the human element. Having a better software process will get you into systems integration in better shape, but no matter how good the software process is, you really won't know how many bugs a system has at the start of integration. Nor will you be able to predict your rate of finding and fixing them, nor how reliable a system may eventually become, nor the time will it take to get there. Standard metrics such as mean time between failures (MTBF) only make sense after systems integration, because before systems integration the system as a system only exists as a concept, that is, the system is an aggregation of software and hardware components with faults sprinkled throughout. The dreadful fact of having to live with poor software fault metrics is kept hidden, much like the mad aunt kept in the attic that no one wants to talk about.

Added to the challenge in estimating faults is the challenge of isolating any one particular fault. The larger, the more distributed, and more complex the system, the more difficult fault isolation will be. Because of design, cost, and test philosophy tradeoffs, system software intended for hardware fault isolation (built-in test) will be limited. In practice, built-In test rarely covers better than 90% of the digital portion of any subsystem, rarely covers better than 90% of the subsystem interfaces, and may be non-existent for subsystem analog paths or components. From the software perspective fault handling routines are often lightly tested and may themselves have faults. Difficulties in fault isolation are compounded by the physical, logical, and temporal distance between the visible error the fault produces and the fault itself. Any perceived error can be the last link in a causal chain of faults, where fixing the visible effect simply exposes the next in the chain. Manual fault isolation is a frustrating, time consuming effort. There will be transient faults, where repeating the known fault conditions will not repeat the error. There will be Byzantine faults, where a system mimics a working system, but really is not. There will be Heisenbugs, where the effect of adding code to unmask a fault causes the fault behavior to disappear or mutate. In a large system the systems integrator performs manual fault isolation by monitoring the various subsystems internal status, and state as the system progresses from event to event. The various status and state elements can be ordered into a sequence of what happened (or didn't happen) in response to specific stimuli. If a visible error does not leave a trace in status pointing back to a fault, then the subsystem software will need to be modified to provide additional status. When software modification is needed across more than one subsystem, coordination and negotiation is needed across the various subsystem stakeholders. The negotiation can take more time than actually changing the software. And once the fault is isolated, negotiation and coordination may again be needed for the correction to be made. Design cleanliness may be sacrificed to political expediency.

The subsystems of a system must be up and stable before the system can be debugged. For large systems there can be an interval on the order of tens of minutes from the individual subsystems power on to when system is ready. Critical faults that occur during this interval will require another power down - power up cycle. When integration is underway, the system may not stay up very long; perhaps on the order of several minutes. The disheartening result can be that in any given day, more time is spent waiting for the system to be ready than in debugging. Something to consider, managerially speaking, is that while integration progress is measured by proving functionality (or "threads of control") across subsystems, perhaps, given the nature of the system, the effort would be better served if redirected towards increasing reliability. Hold this thought.

SOFTWARE TOOLS

Working backwards from the integration phase to the design phase one can try to identify better tools and techniques to serve the integration of the next system. Improved computer languages alone will not solve the problem. Many computer languages already provide features to support fault handling, but language alone cannot recover from faults when used in a non-redundant distributed system, when a distributed processor runs away, or when the operating system handling the fault crashes. Something is needed outside the computer language. Better debugging tools alone are not enough. If the system crashes, the debugger on the same system will also crash. At present the available tools for systems integration do not yet fill the needs for debugging large systems. The challenge can be viewed as one of needing to unravel design encapsulation from the highest systems level all the way down through the subsystems to the individual processors and processes within. The complexity of the heterogeneity, distribution, and communication in the small scale all combine to prevent just about any mechanism for coordination on fault detection and recovery in the large scale. When faults occur, there is the need filter and work the causal chain to determine to who faulted first and to determine correlation across multiple concurrent perhaps independent faults across subsystems. Language and tools are not enough. Improving the development process is not enough. Can there be a systems architecture that, by design, properly supports large systems integration?

SYSTEMS ARCHITECTURE

The subsystems may each behave properly individually, but when connected together system faults can still occur, irrespective of good architecture and design and practices. In the typical systems architecture, the physical subsystem interface is the firewall that is intended to prevent fault propagation across subsystems. In practice, the subsystem firewall is easily breached by the communications software that connects the subsystems. Middleware is both a breach in the firewall that can propagate faults and a critical weak link when the middleware software itself fails. Other sources of faults that propagate across subsystems include global names, subsystem configuration, and coordination and timing of sequences of commands. In some instances portions of subsystem interfaces can remain untested until systems integration. For any given pair of subsystems that have a common interface there is the possibility that each has not been tested exactly the same way on its side of the common interface. Subsystem testing, no matter how extensive cannot replace the testing provided by systems integration.

One of the greatest challenges in integrating large systems occurs when concurrent processes distributed across subsystems raise concurrent exceptions. There are few tools for modeling distributed processes or for modeling fault propagation across distributed processes. Without these models there will be challenges in improving our predictive knowledge of large system behavior. Very simple models for fault tolerant software components do exist. A model of an idealized fault tolerant component has two features, the first is to perform system services, and the second is to provide a fault handling. The fault handler, on fault detection, recovers or propagates status back to the caller. A system built out of components (fault tolerant or not) will only be as reliable as its weakest serial component, that is, the component with the poorest fault handling capabilities. For an immature application, the weakest link is the application. For a mature application, the weakest link is the software infrastructure, that is, the COTS, the operating system, the middleware. How does the generic large system behave in the presence of faults? Faults that are expected are detected, logged and status is propagated to the user. Faults that are not expected are not detected and can cause immediate failure or can remain to build up in the system like a trail of closely spaced dominoes waiting for a triggering push. Some systems are regularly intentionally rebooted to preempt the triggering of accumulated faults; your PC, for example. Preemptive rebooting is one way that humans can cope with systems that are too complex to be made reliable. But preemptive rebooting is admitting defeat. Someone, somewhere, has given up, with the rationale that it is just not cost effective to isolate and remove any more software bugs.

COORDINATED ATOMIC ACTIONS

Is there anything that be done to improve the status quo? One concept that appears promising for systems architecture is Coordinated Atomic Actions (CAA). CAA is a concept that organizes software to provide more effective fault containment across distributed systems. CAA was first invented by members of Brian Randell's research group at the University of Newcastle at Tyne in the mid 1990's. The concept is simple, but the terminology used in CAA requires some explanation.
- An *Action* is an abstraction that allows the systems application programmer to group a set of operations into a single logical execution unit.
- A *Multiparty Action* in an action where several processes (parties) cooperate to produce an intermediate combined state, exercise some activity, and then leave this interaction and then continue execution independently.
- An *Atomic Action* is a Multiparty Action that allows for exception handling for cooperative error recovery.
- A *Coordinated Atomic Action* (CAA) is a Multiparty Atomic Action that also provides for controlled access to shared external (transactional) resources. In CAA, transactions are applied to actions just as database transactions are commonly applied to data.

The steps to implement CAA are simple:

1. Synchronize the participant processes upon entrance and validate the input parameters.
2. Perform the intended functions in order. Handle exceptional conditions locally.
3. Wait for the whole interaction to finish or timeout.
4. Assert post conditions on output parameters.
5. Resynchronize. If something has gone wrong locally, let all participants know success or failure.
6. Complete transactions on external objects. Prevent intermediate results from being passed. If a fault could not be handled at a lower level, handle now.

A software API for CAA could be implemented with a function call for each of the six steps listed above:
1. synchronizeBegin ( )
2. preCondition ( )

3. actionExecute ( )
4. postCondition ( )
5. synchronizeEnd ( )
6. externalUpdate ( )

Any implementation would involve basic concepts known to computer science majors: process monitors to determine the liveness of the processes involved, inter-process messages to synchronize and coordinate the actions of the distributed processes, and semaphores for synchronization and mutual exclusion. CAA designs interface tests explicitly into the subsystem architecture by the use of preconditions and postconditions. Systems integration challenges such as ordering of command sequences and message timing are made explicit and encapsulated within the architecture within actionExecute. As one person's system is another person's subsystem, another advantage to CAA is that the architecture is "nestable". The CAA technique can be hierarchically applied to scale ever larger systems, and flowed down as requirements to encapsulate subsystem suppliers. Nestability permits CAA to encapsulate legacy systems that do not support CAA and cannot be modified.

CONCLUSIONS

Systems of Systems have a disproportionate number complex integration challenges. The holistic view of the system is weakened as subsystems become more independent. Providing an increasing number of controls over the development process may not provide as cost effective a return as improving system design architecture. Advances in systems architecture hold promise in dealing with complexity if the systems view can be maintained all the way down through the abstraction and encapsulation. Payback for improved systems architecture occurs during systems integration. Systems integration is measured by the accumulated integration milestones achieved per successful thread-of-control tested. In traditional systems design, there is little architectural support for debugging faults that cross subsystem boundaries. Coordinated Atomic Actions is an alternative architecture that logically partitions large systems into fault tolerant threads-of-control. With CAA the system boundary is moved from the physical subsystem to the logical thread-of-control. CAA scales with system hierarchy and provides debugging hooks across subsystems. CAA provides a linking concept between fault tolerant systems level architecture and the current challenge of integrating a SoS. Now for the "A-ha!" moment I promised at the start of this paper - The CAA encapsulation of the thread-of-control is the very same milestone used as the metric for systems integration progress! The result and benefit of using CAA for the systems integrator are threefold. One, that systems understanding is increased within the architecture itself. Two, that the debugging effort is decreased for the systems integrator as more of the implicit is made explicit. And three, the integration effort that is performed is the same as the effort that is measured. Accepting anything less in a system's architecture would simply be accepting more of the same.

ACKNOWLEDGEMENTS

I would like to thank Paul Lee for his advice and support shepherding this paper through the approval process. I also would like to acknowledge the inspiration acquired from working with Danielle Ericksen, Beth Finn, Barry Furnival, Dave Giavonnelli, Craig Hanna, Bob Jones, John Mackanic, Mike Narkis, and Rick Schulz on a difficult systems integration; heroes one and all. And last, but not least, Alexander Romanovsky at the University of Newcastle, for providing key CAA papers that were published out of my "electronic" reach.

REFERENCES

Very little attention has been paid to CAA in the US, most likely due to the Not Invented Here (NIH) mind set. Many papers on CAA, dependability and reliability can be found on the IEEE website: http://ieeexplore.ieee.org/Xplore/DynWel.jsp

“On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems” Beder, D.M., Randell, B., Romanovsky, A., Rubira, C.M.F., 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Margeburg, Germany, May 2001, pp. 103-112

“High-Availability Computer Systems”, Jim Gray, Jim, and Siewioreck, Daniel P., IEEE Computer, September 1991, pp. 39-48

“A Distributed Object-Oriented Framework for Dependable Multiparty Interactions”, Zorzo, A.F. and Stroud, R.J. In Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, (OOPSLA ‘99) Denver, Colorado, pp. 435-446

http://www.cs.ncl.ac.uk/people/home.php?name=alexander.romanovsky