Raging Incrementalism: Harnessing Change with Open-Source Software

John C. Georgas
Institute for Software Research
University of California, Irvine
Irvine, CA 92697, U.S.A
+1 949 824 5160
jgeorgas@ics.uci.edu

Michael M. Gorlick
The Aerospace Corporation
P.O. Box 92957
Los Angeles, CA 90009, U.S.A
+1 310 336 8661
gorlick@aero.org

Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, CA 92697, U.S.A
+1 949 824 6429
taylor@ics.uci.edu

ABSTRACT

Change is a bitter fact of life for system developers and, to a large extent, conventional practices are aimed at arresting change and minimizing its effects. We take the opposite view and are exploring system engineering practices that harness the forces of change for the ongoing, incremental improvement of systems—a view we name *raging incrementalism*. We harness three powerful forces to ride the waves of change: open-source software, commodity hardware, and web-like, representational state transfer architectures. This paper describes an early experiment in applying raging incrementalism to a complex system: large-scale digital video capture, distribution, and archival for launch range operations. We outline the methodology of raging incrementalism, describe the vital role open-source plays in system development and construction, and offer insights on the programmatic consequences of embracing open-source software.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*life cycle*

General Terms

Design, Experimentation, Management

Keywords

Raging incrementalism, open source, representational state transfer

1. INTRODUCTION

Change is inescapable in computing systems appearing as shifting requirements, the evolution of deployment environments, and ongoing improvements in the software and hardware building blocks that compose these systems. Most technologies relevant to computing—such as storage capacity or the speed of computation—exhibit a doubling period over which time the performance/price ratio for that technology doubles. While the exact span of doubling periods varies with the technology under consideration, they yield exponential improvements in technological capabilities.[1] Further examination of these doubling periods, however, reveals that they themselves are shrinking; these two factors combined—exponential improvement and a shortening of the doubling periods—lead to a hyperexponential rate of technical improvement [6].

In our specific domain of interest—launch range operations for space vehicle launch—change of this pace and magnitude calls into question all of the accepted canons of system engineering and development. Conventional practices, such as early commitments to specific hardware configurations and "freezing" requirements, are aimed at arresting and minimizing the effects of change, often resulting in systems that are obsolete long before they are delivered.

Raging incrementalism is a system engineering methodology that accounts for change by embracing it in all aspects of system design and development. The methodology relies on three key principles to create scalable systems capable of absorbing ongoing change: open-source software, commodity hardware, and explicit architectural models based upon REpresentational State Transfer (REST). Raging incrementalism leverages the resources of the "creative commons" by exploiting open-source software at all system levels and confines software development solely to domain-specific software for which there are no available or suitable open-source alternatives. Commodity hardware platforms are easily deployed, readily available, inexpensive, and continually improving. Finally, "RESTful" systems can be cast as highly decoupled peer-to-peer architectures, an architectural style that eases the task of integrating heterogeneous open-source with domain-specific components.

Surfing the wave of change leads to profound alterations in the system development process as we shift the emphasis from planning and development toward integration, experimentation, rapid prototyping, and continual system refinement. To this end, we propose a process centered on

[1]For example, the doubling period for disk drive capacity is approximately 15 months, the doubling of transistor density on integrated circuits occurs roughly every 24 months, and the doubling period for the price capacity of flash memory is less than 18 months. Other disciplines, however, may exhibit doubling periods measured in years or even decades.

Open Source Application Spaces: Fifth Workshop on Open Source Software Engineering (5-WOSSE) May 17, 2005, St Louis, MO, USA.

naturalistic architectures that promote flexibility and ease-of-integration: these are architectures that are shaped to accommodate the available open-source components rather than forcing the components to match the expectations and demands of the architecture. In addition, program management must account for a fundamental shift in goals, moving away from custom development toward ongoing integration.

To gain experience with raging incrementalism we developed a prototype digital video system for launch range operations. Constructed entirely from open-source software and commodity hardware, the RAnge Video Experiment (RAVE) served as testbed and "exploritorium" for the efficacy and utility of raging incrementalism.

The remainder of this paper is organized as follows: Section 2 offers background information on raging incrementalism and its principles, while Section 3 outlines the consequences of its adoption from the perspectives of software architecture, development processes, and programmatics. The architecture, components, and domain background of our experimental system, RAVE, are described in Section 4, while concluding remarks appear in Section 5.

2. BACKGROUND

Raging incrementalism [5] is a system engineering methodology that relies on the use of open-source software, commodity hardware, and REST-based software architectures.

2.1 Open-Source Software

No single group of developers, no matter how talented or energetic, is capable of matching the development pace set by the open-source community. The reach and quality of open-source software has increased tremendously over the past decade and we speculate that open-source software, like other technical domains, is experiencing doubling periods in which its performance/price ratio (where performance is measured in terms of function points and price is measured in terms of labor-hour equivalents) sees exponential improvement (though those doubling periods must be measured in years). We also suspect, though do not attempt to prove at this time, that open-source doubling periods are shrinking as well. If that is the case, then open-source software is undergoing hyperexponential improvement. However, irrespective of the precise pace of improvement, there is an astonishing wealth of open-source infrastructure, middleware, libraries, and applications. Organizations seeking to reduce development costs or shorten schedules can ill afford to ignore these resources.

2.2 Commodity Hardware

Raging incrementalism relies upon inexpensive and high-performance commodity hardware for general-purpose computing platforms and deliberately eschews custom hardware solutions. Relying on fungible hardware speeds procurement, eases deployment, and simplifies hardware upgrades. The unprecedented availability and low cost of commodity computing hardware implies that processing and data storage—for the domain and scale of applications that concern us—cost so little as to be practically irrelevant. Furthermore, many challenging system design problems can be economically attacked and resolved with sheer brute force. For many launch range systems the hardware costs, as a fraction of total lifecycle costs, are in the noise; software, personnel, and maintenance costs come to dominate launch range budgets. From this perspective, commodity hardware that substantially reduces overall system costs is a wise investment.

2.3 Software Architecture

Emphasizing the rapid, flexible, and ongoing integration of open-source components as a substitute for expensive, custom-built software, raging incrementalism rests heavily on the system equivalent of "naturalistic architecture," a school of building design that exploits—whenever feasible—the building materials found in the local environment. Representational state transfer, explicated as the design principles underling the modern web [4], emphasizes decoupled compositions of independent computing elements communicating via the exchange of resource representations. REST is uniquely suited to incremental development, rapid change, and the integration of heterogeneous components as it accommodates disconnected operation and fluctuations in resources and membership while maintaining transparency and decoupling in component interactions.

3. IMPACTS AND CONSEQUENCES

In this section, we briefly explore the architectural and programmatic consequences of raging incrementalism. First, our system designs are centered on continually evolving naturalistic architectures crafted to promote flexibility and ease of integration. The system process must account for the infusion of open-source infrastructure and components, commodity hardware, and a continual process of experimentation, refinement, and redeployment. The measures of program success must change to account for the changes in focus: integration versus custom code development, open-source evaluation and comparison versus the evaluation and comparison of closed products, and continuous change versus frozen requirements and specifications. Finally, even the staffing requirements for raging incrementalism differ from those for conventional projects.

3.1 Software Architecture

Raging incrementalism is distinctive in its emphasis on naturalistic architecture—system design driven by the open-source "building materials" that are at hand. Oftentimes in conventional practice, the system architecture precedes and frames component development; in other words, the software components are *built around the architecture*. In contrast, raging incrementalism reverses this order—open-source component discovery and evaluation prefaces and frames the architecture. We liken open-source software to naturally available building materials found in the local environs and insist that the architecture draw from, and depend upon, these materials (components). This ordering is the software equivalent of the school of naturalistic architecture, a philosophy and practice of building design and construction that emphasizes the use of local, indigenous resources (wood, stone, or adobe for example).

Raging incrementalism emphasizes that developers are better able to pace technical progress by developing "early and often." Projects in which long periods lapse between design, implementation, and deployment are more likely to deliver obsolete products ill-suited to the needs of the customer. Since time is precious and resources (money or personnel) scarce, the forces of hyperexponential improve-

ment suggest that developers avoid wastefully reinventing or redeveloping infrastructure for which adequate open-source implementations are readily available. Far better instead that the skills and resources of a team be devoted to the narrow and specific elements of their problem domain for which no adequate open-source solutions exist.

In the world of open-source software development, the chances are excellent that someone else, somewhere else has already solved your problem, in which case, it is a foolish waste of effort to solve it again. On the other hand, it may easily be the case that no adequate or suitable open-source solution exists for some vital and critical element of your problem. Therefore, developers must organize their architectures and processes to establish a clear and bright delineation between those elements of the problem domain for which open-source components are available and adequate, and those for which a custom implementation is required. Our experiences suggest that developers often grossly misjudge the extent to which their application requirements are "special" or "unique." Critical analysis often reveals that the vast bulk of their requirements are neither unique nor even worthy of special consideration, and are easily satisfied by widely available, low-cost, components and infrastructure.

Naturalistic architecture does not dictate the form of a structure or its purpose (however, the materials themselves may strongly bound the structure as in, for example, the limited height of a wall constructed from adobe)—it simply suggests that builders give first consideration to the environment and materials at hand. There is an obvious tension between architecture and materials, as effective architectures rely upon the properties of the underlying materials. However, in our view, the value of the materials (here open-source components, libraries, and infrastructure) outweighs the niceties of architecture and, where required, architectures may (and should) be bent to either accommodate or ameliorate the limitations of the materials.

The open-source components in hand may well, as a consequence of their designs, implementations, or interfaces dictate numerous, fundamental system interactions, again requiring that we *architect around the available materials* rather than shaping expensive (that is, custom-built) materials to fit the architecture. Under raging incrementalism, most of the labor is expended in integrating existing components rather than constructing new ones.

Since integrating diverse open-source software elements is the basic order of the day, we are best off adopting compositional mechanisms that support flexibility and heterogeneity. REST, a protocol-centric architectural style based on stateless interactions, permits combining components whose implementations and forms of interaction may vary widely in kind, scale, and frequency. Raging incrementalism rides on the back of architectural models that are maintained throughout design, development, and deployment—bringing sanity to a process (sketched in Figure 1) that, by definition and design, is subject to constant refinement and revision.

3.2 Development Process

While conventional launch range development practices include elements of parallel design and development activities, these processes emphasize the early binding of design decisions and premature commitments to deployment

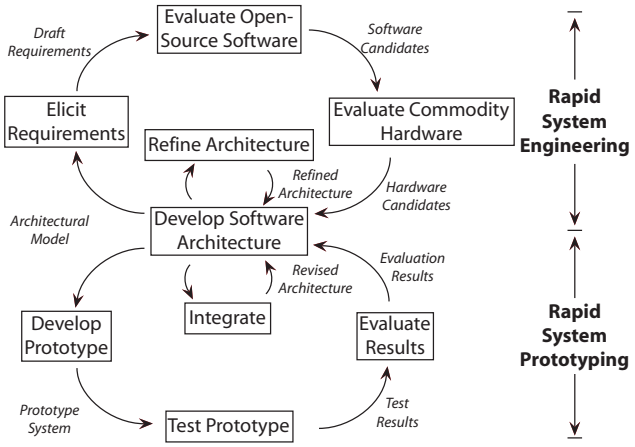

Figure 1: An overview of our proposed raging incrementalism development process illustrating coarse-grained process steps as well as their inputs and products.

specifications. Unfortunately, these decisions are often made far too early in the system lifecycle—long before developers have a clear view of system needs and growth. This, coupled with multi-year development cycles, often results in systems that are inflexible and outdated soon after (if not before) their delivery and deployment.

Raging incrementalism, in contrast, delays binding decisions for as long as possible while explicitly retaining the flexibility to take advantage of changes in hardware, components, and architectures. The overarching goal is to minimize, to the fullest extent possible, the amount of project- or domain-specific software development, reserving such development for those absolutely critical portions of the system whose requirements can not be met by any open-source offering.

A process model for raging incrementalism is illustrated in Figure 1. In spirit, it resembles the spiral development model [3], and iterates through a core set of steps that are centered on an explicit architectural model; the two subprocesses, with the software architecture in the center, may be pursued in parallel.

The top outer-half of Figure 1 defines a *rapid system engineering* (RSE) process concerned with the evaluation, selection, and composition of open-source components and commodity hardware. Beginning with the elicitation of a set of draft requirements (given the current understanding of the system) the process continues with an examination of the available and relevant open-source offerings and identifies candidate packages; the primary aim of this selection step is to identify components that meet as many of the system's functional requirements as possible. The next step, informed by the requirements of the open-source software selected in the previous activity, examines and identifies available commodity hardware best suited to the requirements. The sub-process concludes with the elucidation of a draft system architecture that incorporates and integrates the candidates from the list of open-source components identified while also keeping in mind the performance characteristics of the available commodity hardware.

The bottom outer-half of the process illustrated in Figure 1 defines a *rapid system prototyping* (RSP) effort whose focus is the rapid development and evaluation of prototype systems. Using the architectural model as a basis, a prototype is quickly constructed (construction times must be on the order of weeks to months at most). It is important to note that this system prototype may well be significantly more complete than conventional early prototypes since open-source components may easily span the vast bulk of the required functionality. The system prototype is then tested and evaluated and the outcomes of these two activities guide the refinement and revision of the system architecture.

Further iterations of either sub-process begin using the architectural model as a starting point. The results of rapid prototyping, improvements in open-source offerings, or the appearance of new open-source components may stimulate a radical "rethinking" of the architecture, the software constituents, and the hardware platforms, which may require architectural revisions and refinements (illustrated in the inner cycles of Figure 1).

3.3 Programmatics

While the programmatic consequences of raging incrementalism are still unclear, early experience provides us with some insight on two issues: productivity measures and project staffing.

The emphasis raging incrementalism places on the use of open-source software has significant consequences for personnel productivity metrics. Conventional methods for measuring productivity are based on source code artifacts such as lines-of-code or function points [2]. While reasonable in contexts where there is significant internal project- or domain-specific development, these measures are useless and irrelevant in a development process whose principal goals are crafting system architectures and integrating off-the-shelf, open-source components. As an alternative we suggest that productivity be measured, and project milestones be set, in terms of the number of integrated components, thereby encouraging a focus on system quality and function rather than the quantity or size of the artifacts employed in the process.

Embracing raging incrementalism also places novel requirements on project staffing. Superlative programmers, while still important, are no longer the most desirable members of a development team. Since significantly more effort is expended on architecture and integration, program managers must recruit skilled architects and integrators. These tasks require a deep knowledge of a variety of programming languages, a strong sense for and knowledge of architectural styles [7], and the ability to quickly grasp the important features of multiple components. Moreover, the need for a special role becomes apparent: an *open-source software surveyor*. This is a software engineer who concentrates on staying current on the latest developments in the open-source community and has a deep knowledge and understanding of the capabilities and limitations of particular open-source software packages as well as potential interpackage conflicts—critical knowledge during both development and deployment.

4. EXPERIMENTATION

To better understand the consequences of raging incrementalism and open-source-centric development, we experimented with the construction of a wide-area digital video collection, distribution, and archival system: the *RAnge Video Experiment* (RAVE). The following sections discuss the characteristics of the problem space, the RAVE architecture, our experience with the system's development, and lessons we learned.

4.1 Problem Space

Vandenberg Air Force Base in Lompoc, California hosts the Western Launch Range: approximately 100,000 acres of space vehicle launch pads and missile testing facilities and the only United States site suitable for polar orbital launches. Launch range operations are monitored through a collection of hundreds of analog video cameras. These cameras are essential tools for range operations including launch preparation and monitoring, launch vehicle engineering, post-launch analysis, and physical security. The system in place at the Western Range requires an enormous, circa-1984, video switch and a dedicated cable plant. Cameras, video switch, and cable plant are all obsolete and increasingly difficult to repair and maintain.

The RAVE system, a prototype replacement for the range legacy video infrastructure, aims to provide an all-digital video system that matches or exceeds, in every respect, the performance of the system now in place, while simultaneously reducing maintenance costs and removing barriers to upgrades or reconfiguration. To this end, we applied the principles of raging incrementalism and developed a prototype system entirely from open-source software products and commodity hardware.

The majority of video cameras on launch ranges are black and white security cameras and their performance characteristics were adopted as the benchmark for our prototyping. To match the current solution, our prototype system was required to:

- Produce full color, 320×240 video at a minimum of 15 frames per second
- Generate high-quality, standards-compliant, compressed video
- Scale to hundreds of individual cameras

Additionally, our system would add significant additional, new functionality including:

- Remote network camera control for on/off, focus, color balance, frame rate and other characteristics
- Support for heterogeneous video clients ranging from desktop hosts to terabyte-scale archives
- Arbitrary video switching, with every video camera accessible by any client

4.2 RAVE Architecture

The overall architecture of the RAVE system is illustrated in Figure 2. The system is composed of independent and decoupled *bricks*: individual host machines dedicated to a single function, or encapsulations of a family of closely-related functions. For example, a RAVE camera brick is a dedicated host running only the software necessary for video capture. Isolating system functions in bricks imposes system modularity, increases reliability, and eases upgrade and repair since it is a comparatively small matter to replace one brick with another.

The RAVE system contains four kinds of bricks: *camera*, *proxy*, *streaming*, and *archive*. Camera bricks are small

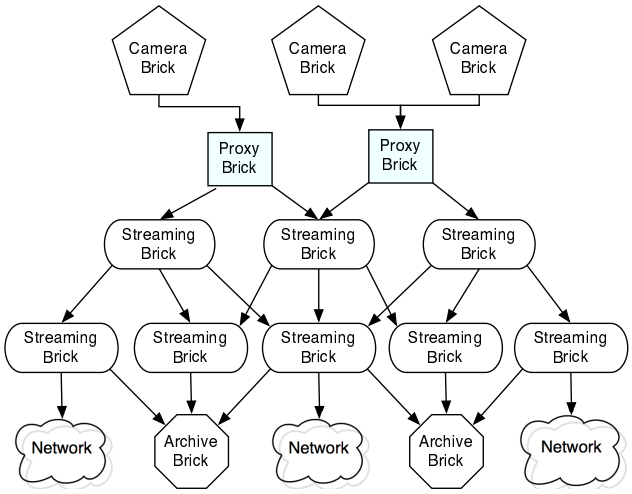

Figure 2: The architecture of the RAVE system, consisting of a peer-to-peer construction of components for video capture, stream proxying and rerouting, multicast streaming, and archival.

form-factor hosts connected to industry-standard, Firewire-compliant cameras [1] and generate MPEG-4 digital video streams using a software codec. Proxy bricks are protocol bridges that insulate camera bricks from the details of communication with streaming bricks. Streaming bricks are high-performance hosts collecting multiple unicast video streams from proxy bricks and performing multicast dissemination of those streams to clients. Finally, archive bricks are specialized clients (namely, terabyte-scale video archives) that subscribe to, and record, video streams for review at a later date. Additional clients, such as desktop video players, may subscribe to video streams by connecting to streaming bricks.

4.3 Open-Source Software

Open-source digital video relies upon a complex, diverse collection of software components, and the RAVE system depends entirely on a number of open-source software packages for its constituent bricks; we itemize the most important of these:

- *Debian Linux* and *FreeBSD* are the brick operating systems
- *libdc1394* is a library for IIDC-compliant Firewire cameras used in the camera brick
- *spook* is a video broadcaster and camera control application deployed on the camera brick
- *xViD* is an MPEG-4 codec used by the camera brick to generate MPEG-4 encapsulated streams
- *mencoder* is an MPEG-4 encoder and container generator deployed on archive bricks to create video recordings
- *Darwin Streaming Server* is the backbone of streaming bricks for video distribution
- *mplayer* is used for video playback by client hosts accessing either video streams or recordings
- *MySQL* is an open-source relational database used by archive bricks for indexing video recordings and associated metadata
- The *live.com* RTP/RTSP library is used for the management of media streams by proxy bricks (in addition to providing RTP/RTSP support for mplayer and mencoder)
- *Mozilla Firefox* is used as the web front-end for the network-based camera control user interface
- The RTP, RTSP, HTTP, TCP, and UDP protocols are used throughout for streaming and control

The large number of complicated open-source packages used by RAVE imposed a steep learning-curve—the library dependencies were particularly troublesome–however, the effort was far less labor-intensive than that which would have been required to implement this functionality from scratch. In many cases, lack of adequate documentation (the Darwin Streaming Server and MySQL are two notable exceptions) was an impediment to swift and painless integration. In general, the poor quality and scope of open-source documentation is perhaps one of the most substantial barriers to the widespread adoption of open-source software.

4.4 Commodity Hardware

Raging incrementalism dictates that commodity hardware be used where possible to reduce costs, simplify procurement procedures, ease maintenance, and promote system longevity. The deployment platforms for the RAVE system are all commodity hosts purchased over the web.

The RAVE video cameras were low-cost Firewire cameras conforming to the IEEE 1394 standard [1]; a wide range of cameras are available and Firewire ports are commonplace on commodity computing platforms. Camera brick hosts were small form-factor Shuttle platforms,[2] while streaming servers and prototype archive bricks were hosted on server-grade hosts.[3] The video archive bricks are 4U rack mount servers containing approximately four terabytes of RAID storage. They were assembled to our specifications by a server vendor a few years ago and are constructed entirely from widely available commodity components. At the time of purchase they cost approximately $6000 each, but comparable "data bricks" can be assembled for considerably less now.

4.5 RAVE Performance

As of this writing RAVE exists as a laboratory system with several cameras deployed on The Aerospace Corporation campus. The camera bricks are extremely robust and run for weeks at a time unattended, but would have to be repackaged in waterproof enclosures before we would consider deploying them to the range. In addition we would like to reduce their size, power requirements, and heat output. We are seriously considering re-hosting the camera bricks on a Mini-ITX single-board computer or cannibalizing an Apple Macmini.[4] The camera bricks generate 30 frames/second MPEG-4 video, twice the rate called for in the requirements, at a (modest) bandwidth of approximately 200–300 kbs per camera. Consequently, we intend to explore increasing the frame size to 640×480 at the cost of a decreased frame rate and increased bandwidth consumption.

[2] `http://us.shuttle.com`

[3] Adequate 1U rack-mounted servers can be purchased online for well under $1000.

[4] `http://www.mini-itx.com` and `http://www.apple.com/macmini`

Our primary complaint is that several components, including the Darwin Streaming Server and the `live.com` RTP/RTSP library employed by the proxy brick insist on buffering the video stream and introducing needless delay between the camera and client. While the delay is irrelevant for some clients (such as the video archive brick), the long delays (on the order of 7–10 seconds in some circumstances) violate range requirements for some applications. We are considering replacing the Darwin Streaming Server with a custom-built server for the streaming bricks, which would allow us to dispense with the proxy bricks altogether, simplify the streaming brick, and eliminate the superfluous buffering. The Darwin Streaming Server would still be useful in some range applications, for example, as a service for replaying video clips stored on the video archive.

Finally, there is the question of transferring a prototype to the Western Launch Range for test and evaluation. The Spacelift Telemetry Acquisition and Reporting System (STARS), developed by The Aerospace Corporation and deployed at the Western Range and at Aerospace headquarters, monitors and analyzes the real-time performance of launch vehicle systems and their payloads before, during, and following launch. We are investigating integrating the RAVE prototype with STARS on the Western Range to augment the analog video feeds from the launch pads.

4.6 Lessons Learned

Development of the RAVE prototype was performed in parallel following the process described in Figure 1. Each brick was developed independently before final integration and testing; a task made comparatively simple by a RESTful architecture that supports strong component decoupling. The *entire* development process occupied one full-time and one part-time developer for a period of six weeks. This is an amazingly abbreviated development period for any complex system; though only a prototype in scale, the RAVE system offers more functionality than the legacy system it is intended to replace at a fraction of the cost. If nothing else it is a graphic demonstration of the power and convenience of open-source software and commodity hardware. Architecture-centric integration demands a different set of skills than does "clean slate" development, and to some, may seem overly complex and difficult. However, it is one of the few practices capable of pacing accelerating technological change.

5. CONCLUSION

Unrelenting change is a defining feature of the modern technical landscape. By ignoring change developers risk wasting effort building systems that are obsolete long before they are delivered or deployed. Raging incrementalism is an architecture-centric system development methodology relying upon open-source software, commodity hardware, and RESTful, peering architectures. Raging incrementalism embraces change at all points in the system lifecycle and makes every effort to exploit change rather than shunting it aside or attempting to minimize its effects.

Launch ranges are challenging operating environments with stringent requirements for safety, availability, and reliability. We believe that, for selected application domains, open-source software offers exciting possibilities for radically shorter development times at substantially lower cost when compared to traditional methods of custom system development. RAVE is one in a series of prototypes for launch ranges that demonstrates the utility and promise of open-source software, commodity hardware, and architecture-centric integration.

However, raging incrementalism does not come without a price and requires a substantial shift in development practices including planning, management, design, development, deployment, and sustainment. We intend to refine RAVE as a demonstration of the ability of raging incrementalism to pace technical change and improvement. In addition, we are exploring other launch range domains such as launch vehicle telemetry distribution, adaptive telemetry processing, radio frequency monitoring, and security perimeter control.

Change is both inevitable and threatening. We anticipate that large-scale system engineering will be transformed by the adoption of open-source software—a powerful and essential tool for addressing ongoing, unrelenting change.

6. ACKNOWLEDGMENTS

This work was sponsored by the Research and Development Program Office of The Aerospace Corporation.

7. REFERENCES

[1] 1394 Trade Association, Santa Clara, California. *IIDC 1394-based Digital Camera Specification*, version 1.30 edition, July 2000.

[2] A. J. Albrecht. Measuring application development productivity. In *IBM Applications Development Symposium*, pages 83–92, Monterey, CA, 1979.

[3] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[4] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[5] M. M. Gorlick. Raging incrementalism—system engineering for continuous change. In *Proceedings of the 2004 Ground System Architecture Workshop*, Manhattan Beach, CA, March 2004.

[6] R. Kurzweil. The law of accelerating returns. March 2001. www.kurzweilai.net/articles/art0134.html.

[7] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Computer Software and Applications Conference*, pages 6–13, 1997.