HealthManagement
Promoting Management and Leadership

Volume 2 / Issue 1 Spring 2007 - Features

A Unified Systems Engineering Methodology for Trustworthy Systems

Formal models help, but the essence is formalised thinking

Eric Verhulst

is director of the Open

License Society.

More information at

www.OpenLicenseSociety.org

Why Projects Fail

As our systems grow in complexity, so does the software content. And software engineering as such is not yet a fully mastered discipline. According to the nowfamous CHAOS studies of The Standish group, about half of all IT development projects still have serious problems although the success rate has been improving over the last decade.

Today, 15% still fail completely while 35% are over time, over budget or lack critical features and requirements. The reasons remain the same: not enough effort is spent on analysing the stakeholder's requirements and formalising them in implementable specifications. This is the human factor becoming apparent. In software intensive systems, the issue is state space explosion and the ensuing application of Murphy's Law - when things can go wrong, they will. It is just a matter of time. No testing will ever reveal all errors, as testing itself could take longer than the lifetime of the system.

Small is Beautiful

When things get too complicated, it is time to apply the principle of Occam's Razor – named after the 14thcentury English Franciscan friar and logician, William of Ockham: cut away all ballast. From an architectural point of view, this also means that we work with smaller units as trustworthy components. There is then no need to export the full internal state space to the whole system if each component can be accessed with a simple protocol. This should also be reflected in the project planning. Rather than trying to develop it all at once, the project manager should introduce small tasks with short-term milestones and work in iterations by introducing well defined and clean interfaces between the components. This is also what the Standish Group found. The total failure rate went down from 30% in 1994 to 15% today, because projects were split into smaller sub-groups and tasks, while the classical waterfall method was replaced by an evolutionary approach.

Prevention, Rather than Testing and Correction

Another major difference between software engineering and most other engineering disciplines is that software is still largely a craft. This is reflected for example in the fact that mathematical tools, called formal methods in this domain, are barely used unless safety is a critical issue (e.g. for the development of an airplane or medical device). This is a bit astonishing as software can be made perfect if it is running on an error-less piece of hardware. Hardware has bugs that are even visible under a microscope. When software fails, it is either a hardware bug or a software design error. A methodological process can prevent such errors or at least reduce them to a fraction of what is common practice (a few errors for each 1,000 lines of code). Testing is now too often used in the hope of finding such software design errors (erroneously called bugs), while it is mathematically sure that testing will never find all these errors. Testing should be reduced to confidence testing only – just like construction engineers, who do not use testing to find their design errors. How can this be achieved?

A Unified Systems Engineering Process

The key to a successful systems engineering process can be summarised in a single word: consistency. What it means is that is that we must take into account that developing a system is an activity carried out by human beings for human beings, with all their differences and limitations. The major limitation is that we are all individuals speaking our own "language". Often this means that we all use different terms for the same thing, because in each domain the terms have their specific context. This can only be overcome by intensive communication and by defining a unifying "systems grammar". In other words: teamwork by people of complementary skills and defining a common vocabulary for all activities in the systems engineering process. It starts by collecting requirements from all stakeholders. Next comes an activity whereby these requirements are translated in measurable specifications and system components. Today, this also means simulation and modelling to verify that all selected

requirements are consistent and complete. Actually when developing a system, we have to consider the ‘operators’ as well as the ‘environment’ as part of the system under development. Simulation can also help to validate the feasibility. Formal modelling helps in proving that the used algorithms are correct but also that the architecture is right. And finally, when all these design activities are complete (this means after a number of iterations), then we need a runtime environment that speaks the same language. At Open License Society we believe that all systems can be expressed as a set of “interacting entities”; hence we need a runtime environment that supports it. The result is a process oriented programming model. In the ideal case, the source code is generated, not written.

Our Test Case: Developing a Network Centric Real-Time Operating System

In order to test this model, Open License Society used it own approach to develop its own supporting toolset. First, a systems grammar was defined and a Web-based supporting tool for collecting requirements, specifications and architectural elements. This environment was applied first to the development of OpenComRTOS (OpenCom Real- Time Operating System) as the future runtime layer. Before any line of code was written, its design was carried out from a blank slate while applying the Leslie Lamport TLA/TLC formal model checker (Temporal Logic of Actions/Temporal Logic Checker). For months, a team composed of a systems architect, a software engineer and a formal modelling engineer worked closely together. Starting from a very abstract and incomplete model, several tens of successive modelling stages resulted in a code that was written and running in a first version in just two weeks.

Several lessons were learned along the way. The first is that formal techniques do not really prove that a software works correctly and is wholly error-free. It can prove that the formal models are correct, and hence a matter of correctly mapping the architecture into source code and using a trusted compilation and execution environment. The major lesson, however, was that the approach also results in much more efficient and robust architectures (e.g. the resulting code could be reduced to less than 1 Kbyte for single processor nodes and less than 2Kbytes for multiprocessor nodes). This is an improvement of a factor 10 to 20 versus a similar RTOS (Real-Time Operating System) that was developed by the same architect ten years ago. In addition, the resulting design has several important improvements for security and safety (e.g. buffers free from overflow) and for predictive real-time behaviour.

What are the Lessons and Benefits?

On the engineering level, the lessons are a bit counter-intuitive. All engineers have been educated as specialists with a single goal - to produce better and cheaper products. However, this has been proven to be just an extension of craftsmanship. It is valid when the domain remains very well confined and when new designs are gradual improvements of existing designs. Our mind is good at this because it requires heuristic experience and it provides us with a road of lesser effort.

Today however things change too fast and systems are increasingly complex and have multiple subsystems coming from different domains. To still achieve the goal of efficiency, this requires more than reuse, it requires re-evaluating existing solutions and rethinking them anew, looking from a different perspective. This was, for example, the case of OpenComRTOS - resulting in a dramatic improvement, even in a well-established domain. Formal modelling helped precisely because it allows to reason at a much more abstract level, without reference to the implementation domain. Hence, generalisation is the key. The project also clearly showed the limits of the human brain, at least for systems engineering. It was almost shocking to discover how much our thinking can be biased by what we know and how hard it was to make the switch to more abstract thinking. This is certainly an area that deserves more research.

Finally, the benefits are evident but must be measured in the long term. They come down to having more trustworthy components to build new systems from, but with higher quality (robustness, safety, security). This approach also has the benefit of reducing the development cycle, reducing the failure rates and redesign costs. In the end, the benefits are financial. Less money will buy us more if we not only correctly develop the right stuff, but also if we develop the right system - which is where it all started anyway.

Published on : Sat, 21 Apr 2007