The CM II Process Framework

Joe Farah, June, 2008

The CMII Process grew up in the 1980's as a CM process that has been mostly used for hardware development, and for the most part remains that way today. CM II was introduced by the Institute of Configuration Management (ICM). Thousands of CM II course graduates, across 30 countries, have worked the CM II process into their own corporate processes, especially in the Aerospace and Defence sector. However, it wasn't until almost the turn of the century that a number of PDM tools supporting CM II started to appear.

CM II has as it's cornerstone the belief that hardware parts are not revisioned. Instead, the documents used to define the requirements, including detailed requirements, are revisioned and used to generate the hardware, which is given a new part number. A "Baseline View" is used to show the product baseline in terms of the documents and parts that are added, replaced or removed as part of each Enterprise Change Notice (ECN). If you like, it is a visual delta of the hardware assembly breakdown structure, in terms of the documents used to produce the parts.

Engineering Change Requests (ECRs), arising from customer problem reports and feature requests, result in a technical impact analysis and cost estimates. Based on ECR priorities and planned release schedules, ECRs are allocated to ECN packages, and from there a work plan is used to implement the ECN. This implementation is, essentially, the production of new or revised requirements documents based on the impact of the ECR on the various product documents. Multiple ECRs are typically bundled into an ECN implementation, although minor ECRs, which have very little impact, are "fast-tracked" into small ECN's which are easy to implement and release.

I've spent a lot of time over the last couple of years looking at the CM II process with Rick St. Germain, the Canadian CM II expert, and whether or not, and subsequently how, it applies to software development. The motivation here comes from the fact that hardware systems continue to take on more and more of a software component and the ability to manage the combined hardware/software systems from a single management tool would seem to have significant product and project benefits. Although I've worked on a number of hardware/software projects, I'm not a hardware development expert, so I'll welcome any feedback.

Software is not Hardware

Software is not Hardware. However, there are a number of clear distinctions between hardware and software. Not only do these make process engineering for a hybrid (SW/HW) system more difficult, but the distinctions tend to keep hardware and software teams in their own camps. Hardware teams generally view software as just another component to be loaded into a PROM or some other hardware component suitable for storage. Software teams view hardware as a platform for the software and the target to be managed by the software.

A clear understanding of the differences between the two goes a long way to bridging this gap between teams. In some cases one team is fearful of opening the can of worms the other side is dealing with, and so tends to describe the other side as a neat black box. Each is happy to have their own tools to control their own processes, quite independently of one another - that is, until product delivery has to occur. So it's time to take a look at these differences and then to look at how the CM II process can be adapted for both software and hardware.

1. Hardware costs occur primarily during production (and subsequent maintenance). Software costs occur primarily in design and detailed source code implementation. Once complete, production is a near-zero cost.

2. Hardware is designed to be rigid and to last. You may design certain modules so that they can be easily upgraded, but for the most part, you don't want to be changing board layouts or

having to go into your inventory and retrofit a whole set of existing production parts. Software, on the other hand, is designed to be easy to change. In fact, that's why we have software - so that we can continue to improve the functionality of a product in the field, in a relatively inexpensive manner. It's a by-product that we can also fix problems in the product in a less expensive manner.

3. Because of the flexibility of software, it is very complex. Hundreds or thousands of features are typical in any significant software product. And because of this, software is more easily driven to a need to automate - automate the specification (source code) to product (executable) process; automate the configuration and build process; automate the part identification process (to the point that the team rarely even has to deal with part numbers). Hardware addresses complexity by introducing very rigid standards - pin layout for a given generation of memory; board connector standards to support the back plane; power standards to ensure interoperability between parts; and electonic routing standards to minimize crosstalk. Assembly is typically partially automated - production of chips and boards, but hierarchical assembly into the product often requires very expensive automation machinery or some human interaction, quite common on smaller production volumes.

4. Hardware change is typically (though not always) part-centric. A specific change will cause the production of a new part, or sub-assembly. A Software change quite often spans the system, dealing with data objects, user interface and control logic all for a relatively simple request.

5. Hardware assembly is typically hierarchical over time, with smaller parts and sub-parts being built and tested independently and over time being integrated into the whole. Although software has some of this behavior, it is mostly built entirely at a push of a button. So a new database program is compiled and built based on the changes applied. The testing is done primarily in the context of the completed build, and not module by module until the entire system can be built and tested.

6. A related point is that software testing is done within the entire system by the developers. They make their changes, perform a build operation, test their changes and cycle through that iteration until they are satisfied that their changes meet the requirements that they have been given. In hardware, there may be some simulation tests, but functional testing of something resembling a deliverable has to wait, not only until production, but also until the entire hierarchy is re-assembled with the changed parts in place.

7. The complexity of software, and the ability to change it, combined, imply that delivered software will have errors. Their may be careful testing against the requirements, but in practice it is impossible take into account all of the environmental nuances and combinatorics necessary to eliminate all errors. In fact, it's less costly to build redundant algorithms and better in-service upgrade capabilities, than it is to try to exhaust testing. And this will typically result in higher perceived product quality - assuming the basic quality is already there. Hardware is different. We want to get every possible problem out of the product before we commit to production runs. Product recalls are expensive. They can't be done over the internet like a security patch can. The result is that hardware development is geared toward true zero-problem delivery while software development is geared towards a much more lenient acceptance level.

8. Quantities are a significant part of a hardware bill of materials (BOM). For software, the quantity is always 1. This is because sharing of software functionality does not require replication of the software.

To sum up:

1. Hardware production cost is great; software is near-zero
2. Hardware is designed to last; software is designed to change
3. Hardware is moderately complex with rigid standards; software is extremely complex with looser standards
4. Hardware change is part-centric; software change is function-centric
5. Hardware is assembled over a significant time line; software is typically built at the push of a button
6. Hardware testing must await production and assembly; software testing is done by the developer prior to verification teams
7. Hardware is built (nearly) error-free; software is delivered with a significant number of known errors, and even more unknown
8. Hardware deals with quantities of parts; software always has quantity 1

There are many other differences, some minor and some not so much. But these should give us a basic framework to help appreciate process differences that will be necessary on each side.

Similarities between Hardware and Software Development

All in all, there are likely a lot more similarities between hardware and software development than there are differences. It's the similarities that give us hope for a unified process for hardware and software. We don't need a process that is identical, just one with the same framework so that each team (hardware and software) can understand and work comfortably with the other. Some of the similarities that are significant include:

- Each starts from a set of requirements that have to be tracked and satisfied
- Each has an assembly breakdown structure
- Each has to track problems
- Each has to perform change management
- Configuration management of the as-built products are crucial
- Unique identification of each object, including parts, documents, source code, etc.
- Verification has to be carefully tracked against built products
- Each has roles and work flow associated with it

CM II for Software?

With CM II we have a lot of familar hardware terminology: Baseline, ECR, ECN, CRB (Change Review Board), CIB (Change Implementation Board), Impact Matrix, and so forth. Though there are some similarities, there are enough differences with software terms to scare away software teams. But this need not be the case, and in a hardware/software project, there are strong benefits to using the same tools.

How do we bring things closer together? Well these are a number of concepts that need to be mapped onto software terminology and there are other differences that need to be reconciled.

(1) ECR: Engineering Change Request

An ECR comes from a Customer/Productd Feature Request or a Problem Fix Request. It is an instruction to change the product. Although I have seen a number of software teams use the term ECR, most prefer the terms Feature, Problem Report, Task or similar terms. Software doesn't have it's terminology as well set here as does hardware. ECR is a well known term through most of the hardware world. And there's nothing wrong with adopting the same term in software. It is important in software to track problems distinct from features, as they necessarily run through different processes. I prefer having separate first order objects for the two, but I can certainly live with a more generic object as long as I know the "type" (problem or feature or other).

(2) ECN = Build Record or Notice

An Enterprise Change Notice (ECN) is also known as an Engineering Change Notice. It bundles a bunch of changes to be applied to the product. Quite often these changes are small - change a resistor value. Often there is a slightly different workaround for the existing inventory (e.g. add a jumper wire). Sometimes, individual changes are bundled as complete ECNs to help expedite them through the system. ECNs are applied in sequence to a baseline to move the baseline forward. Typically a number of ECNs are bundled together to provide a formal product upgrade.

In software, the equivalent is the Build record, Build package, Build notice, or whatever you want to call it. The set of changes that are going to be applied to the last build to advance to the next Build. A record of these can be called a "build record" or simply a "build". A sequence of builds is used to move a baseline forward. Typically, nightly builds are integrated and result in the next development baseline. Sometimes a build is done as an emergency "patch" packaging a single change. More often there are a number of changes, depending on how far down the release path things have gone. Typically, a number of Builds are bundled together to provide a formal product upgrade, in terms of a release.

Just like a baseline level can be identified by the ECN identifier, similarly a software baseline level can be identified by the Build identifier. Build notices are ECNs. Frequency and packaging may vary, but the function is the same.

A note of clarification: we can identify baselines by their unique identifiers, but typically, a hardware baseline is identified by its ECN level, and a software baseline is identified by Build number. When we talk about bundling ECNs or Builds into an upgrade, we're really saying that we apply a sequence of ECNs or a sequence of Build increments to the previous release baseline to reach the upgrade baseline. Often a build, by itself, is used to identify the entire baseline. Similarly for ECNs. It is actually preferred to use the same identifier for both the baseline and the increment. The context determines whether we're talking about the increment (i.e. the changes only) or the complete baseline up to and including those changes.

(3) Production Work Authorization vs Automated Builds

One of the things I've had to struggle with in reconciling hardware and software processes was the work authorization. In hardware, work authorizations are explicit and carefully dealt with. A work authorization may be given to do a production run (ie. build the product or sub-assembly or part). Work authorizations are also used for formal testing efforts and for other significant tasks. Work authorizations are always issued to apply an ECN to the baseline.

In software, we authorize the beginning of a new formal testing session. We authorize releases. But apart from that, we don't do a lot of authorizations. The reason is that the production effort is not costly or time consuming. Rather than using work authorizations to apply a Build to the baseline, typically, builds are applied to the baseline automatically overnight. In a sense we authorize the process to automate nightly builds. But individual builds don't require authorizations - they're just expected to appear every morning, for example.

A related item is the allocation of what is going into a hardware ECN or a software Build. This is done carefully in hardware, regardless of where you are in the release cycle, both with respect to content and sequence of changes. In software, typically all changes are accepted into a build early on in the release cycle, and although the process is more selective later on in the cycle, this is usually controlled, not at build packaging time, but at change authorization time. So it's relatively simple to automatically pick up the appropriate set of changes for a nightly build. Only when builds fail does the manual effort have to come into play. One reason this is possible is that software developers are responsible for doing full "unit" or change testing prior to checking in the change into the repository or promoting it to a "ready for build" status. The same is not normally possible with hardware.

(4) ECN vs Change Package (aka Update)

There is a tendency to equate the ECN with a software change package. This is because many ECNs are packaged as single changes and fast-tracked through the system. It also happens because there tend to be many fewer changes per ECN than there are changes per software Build. A change package, however, packages the set of changed objects into a single package for a specific change task. Typcially this is a problem fix, a feature implementation or a part of a feature implementation, that is, an ECR or part of an ECR implementation.

We use the term "Update" as a synonym to change package to identify a task spawned from an ECR. Typically an update will implement the complete ECR, but sometimes, an ECR will be broken into multiple updates.   The CM II process would benefit from the insertion of the "Update" concept so that where an ECR is implemented as a series of changes, it is clear which parts of the ECR is implemented at which ECN/Build level. Rather than a direct mapping of ECN to ECRs, a mapping of ECN to Updates and Updates to ECN(s) allows a more precise expression of what is in an ECN package, and how the implementation work of the ECR has been broken down.

(5) Source Code is an Implementation Specification; executables are parts

One hot topic of discussion in the CM II world is whether a source code file is a "part" or a "specification". Comparing to hardware, just as parts form an equipment breakdown structure or hardware tree, so do source code files form a source tree. So the tendency is to equate source code to parts. A closer look though shows us that source code is a specification, just as there is a specification for silicon layout. Both are specifications and can be used to automate the production of the parts, in once case, the executable, in the other, the silicon. It is the executables, the .DLLs and perhaps some source files themselves (e.g. run-time scripts, run-time data files) which form the parts.

A closer look shows that a deliverable can be an executable, a source code file (e.g. a script), or even a functional specification file (e.g. a run-time menu definition file). So we need to be careful about how we classify things. Source code is always a specification. It may or may not be a part, and most programming code is compiled and linked into executable parts.

A Unified ALM/PDM System

After looking at the similarities and differences between hardware and software and between the CM II process and a typical software CM process, it's easier to conceptualize a single system that can deal with both. The path forward is two-fold. (1) Process - we've looked at a few of the process differences and tried to reconcile a number of factors. There are many more that need reconciliation but hopefully we've seen enough to see our way forward in the big picture. (2) Tools - CM II tools are still in their first generation, usually as an add-on to a PDM system. Although much further ahead than 1st generation software CM tools (largely because of the full product life cycle focus), there is a lot of catching up to software CM tools to be done, as these begin to enter their 3rd generation. But a consolidation effort might work to move CM II tools ahead an entire generation.

A unified system will have many common components:

- An engineering database
- A state flow capability

- A baseline capability
- Configuration Identification and Management
- Change Control
- ECN/Build Definition (preferably using only one of the terms)
- The Requests/ECRs functionality
- The Requirements Management and Test Case/Test Run Result capabilitiies
- Project Management
- Document Management
- Globally distributed development

The next generation of such an ALM/PDM system will indeed be able to consolidate software and hardware development, without confusion and without the need to cross-pollinate data between systems. This will make management easier as a single view of resources, quality, progress, requests, etc. will be possible across the entire project.