Bill of Material

Version:	1.0
Date:	14. March 2005
Author(s):	Jürgen Salecker, Dietmar Schütz
e-mail:	juergen.salecker@siemens.com, dietmar.schuetz@siemens.com

Summary

The "Bill of Material" pattern - abbreviated with "BOM" describes a solution for keeping track of generated software components and valid software component combinations, especially in the case of distributed embedded system development.
The idea is to include automatic generated version information in the final binary which enables the unique idenitification which components had been used to generate the final binary. All version information is collapsed into one hashcode which includes – in addition to a CM reference - the component version id and name, system and component patch level (if applicable), developer id and also time, date and location depending information. In order being able to report and to identify any component combination a global hashcode will be calculated out of all components which do compose the final binary. By storing the global hashcode together with sufficient access methods on the device itself provides methods for device self-validation and component identification.
A software component in this context is not only software, it is anything which has a specific scope of operation and is exchangeable, like: digital reference images, icons, application code (Java, C, Assembler, etc.), bootstrap loader, programmable hardware (VHDL code), system configuration parameters, system diagnostic application, or digital filter coefficients.

Also Known As

Somtimes referenced as a "Manifest" file, however then usual compile time information is not included, nor a digital signature of valid component combination. Some configuration management (CM) systems (like *ClearCase*©/*Rational*© and *CM Synergy*©/*Telelogic*©) do have a similiar concept under the same name "BOM", but then it is coupled with the CM system itself, it also does not include pre-built components and not non real software components like VHDL code, etc. which are mission critical to have in the same scope as "real" software components.

Example

Consider a typical embedded system development environment, where a lot of hardware devices (in different hardware versions as well) are available in the LAB(s), are field tested at different locations and therefore being very mobile. Now add some agile developers, test- and integration teams, who are eager to test new features, bug fixes or enhancements as fast as possible. How can they know (without massive phone conferences, verbal and written communication and the like) which software version and component combinations they are testing and whether the version is a valid (for testing) one? Remember at this time there exists not any official software version or release. The answer is they are able to use the information available in the various BOMs to know exactly which software versions and especially component combinations they are currently testing. One unique digital signature calculated out of all BOMs will be used to either validate a certain component combination or it will be used in addition for communicating, identification and verification of a unique and well-known software configuration.

Context

An agile and distributed embedded system development environment where different software components, or cluster of components do compose an embedded system.

Problem

Traceability and product self-validation – without ineffective administration procedures – are essential cornerstones in an agile development environment. Otherwise questions like, which software version did succeed which test, contains this device a valid combination of software/hardware components, who changed this component, because now the product fails a test which succeeded in earlier phases, could not be answered efficiently and are essential to know for an effective steering of the development and testing team.

In agile development environments which are working at high speed together with a high rate of change, it is a significant challenge to keep track of any intermediate version of software components (including their valid combinations) and being able to know whether an embedded device contains a valid or invalid software component composition.

Even in today’s time it is a fact that sometimes (but still far too often) very basic methods for file transfer into other systems are used. In order to know which file version is shipped, the file version information is embedded into the file name itself. This is easy but also an excellent receipt for creating serious misunderstanding, leading to potential wrong component composition of an embedded system and simply loosing the traceability of a certain software version. Especially the last case can be expensive (in terms of financial resources) if just one specific but unknown version works well, where all other versions fail to work on the embedded device. Let's assume a VHDL based component for the programmable hardware devices has to be shipped to a manufacturing database system. Then it can happen (and happened, of course since Murphy is still alive) that the chosen file name did not fit the constraints defined by this database system (consider just the classical problem maximum eight characters per file name, or not more than one “.”, etc.), then the version information which had been embedded into the file name will be lost with the consequences mentioned already above.

Another problem area is more related to standard developer work style. Assume a developer has finished an enhancement or solved a bug then he needs to test his code. In embedded system development usually the code must be downloaded into the embedded device. Only after he has executed this task he will able to test his code, however in complex environments it is very common that he hands over his code to a test- or integration team, as otherwise the required test coverage is not achievable. This team has the tasks of (1) testing with a different view, (2) performing integration tests and (3) testing in a more advanced environment as the one available to the developers. In extreme complex embedded system developments like systems for oil-exploration, space missions, or other mission critical systems, it is nearly impossible that a sufficient test environment can be made available to every single, a group of developers, or even in worst case to a test team itself. Especially in the last case the testing has to be done under semi-commercial conditions which requires a perfect knowledge of even any intermediate system version, this especially includes any software at any time of every developer work-area. In those environments the test equipment is very expensive due to the required physical space, the monetary resources, a very distributed environment, or a combination of all those items. The problem becomes worse if the embedded device requires a various mixture of software components to be operational. In those cases even the combinations of software components have to be verified and communicated in an automatic fashion.

The following **forces** influence the solution:

- Providing traceability and system self-validation of any especially intermediate software versions especially in the light distributed and separate test teams in agile development environments.

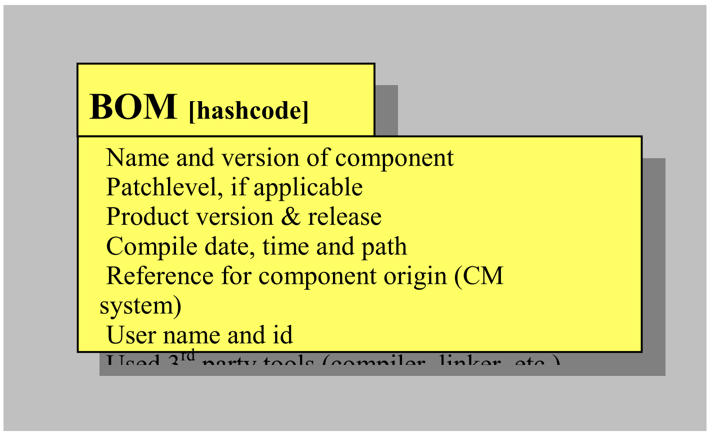
- For embedded system development it is essential to include all components into one common versioning scheme. Components in this context are first the classical ones written in the high level software languages like: Java, C#, C, C++, various assembler languages) and second every other component which is essential for correct system operation. Those non-software components could be provided via VHDL (a programmable hardware language), Verilog (another language for programmable hardware) or a just configuration parameters defined in XML documents, or unstructured third party data.

- Having a method which allows identifying the origin and version of a file without having to worry about thousand different file formats.

- Having a method to verify the version and origin of a file, without the need to trust some kind of limited and "hand" made version information embedded into the file name
- Being able to provide a change for an extreme urgent customer bug fix or some a brief field test direct out of a developers work-area without having to go through a complete integration process. In very agile embedded system developments it is not acceptable that for every field test, or extreme urgent customer bug fixe the required change has to go through a complete integration process.
- Being able to achieve a low granularity for component versioning which relies not solely on a CM system based versioning. A CM based versioning could be far too heavy weight as it usually allows not the control over temporary (we have to identify not yet checked-in components) components which are still subject to potential change.
- For mission critical systems it is necessary to distinguish (the version) by "just" a re-compilation (without any changes in the sources) with a new and automatic generated version id/hashcode.
- Being not able – due to monetary resources required – that a sufficient test environment can be made available for a single developer, a group, or even in worst case the test team itself, then testing has to be done under semi-commercial conditions.

Solution

The basic principle of the solution is that all components which require version information have to be identified. Then the build process of those identified components has to be automated, by taking into account especially non-software components. For every identified component the version information will be collapsed into one version signature, and then a global hash function calculates a global version (device) signature out of the all individual version signatures. The capability for system self-validation can be provided by embedding the generated global signature into the generated system together with sufficient access methods.


BOM [hashcode]
Name and version of component
Patchlevel, if applicable
Product version & release
Compile date, time and path
Reference for component origin (CM system)
User name and id
Used 3rd party tools (compiler, linker, etc.)
Figure 1: Bill Of Material

As a first step towards a solution, all components which compose the embedded system have to be identified.

Identification of Components

This sounds easier as it really is, because it has to include also “non” software components, i.e. anything which has a specific scope of operation. For example, programmable hardware (usually written in VHDL, or VeriLog) is considered in this context as an essential component in the same way as configuration parameters (usually defined in XML, or text strings) and of course all “classical” software components like application code, communication stacks, power management, bootstrap loader, and even the CPU reset vector.

Adapting the Build Rules of the identified Components

The build rules of the identified components have to be adapted for BOM package generation, special attention has to be given to VHDL based components, because those components usually do not allow to embed textual information.

BOM Generation

This generation is itself a three phase process, first the component specific BOM in a generic format (for example XML) is generated, second the generic format is transformed into a component specific format being able to embed this BOM "Figure 1" into the component (which could be in case of VHDL component direct INTEL HEX, or similar formats), other specific formats could be any assembler language, Java, C, C#, or whatever matches with the components core language.

Target BOM Generation

Finally a digital signature is calculated from the generated target/component file and appended to the BOM in the generic format. This – now called – Target BOM is a one-to-one copy of the already embedded BOM and contains in addition just the digital signature of the generated file. The Target BOM "Figure 2" has to be kept at a central place, i.e. the CM system therefore allowing back referencing at any point in time.

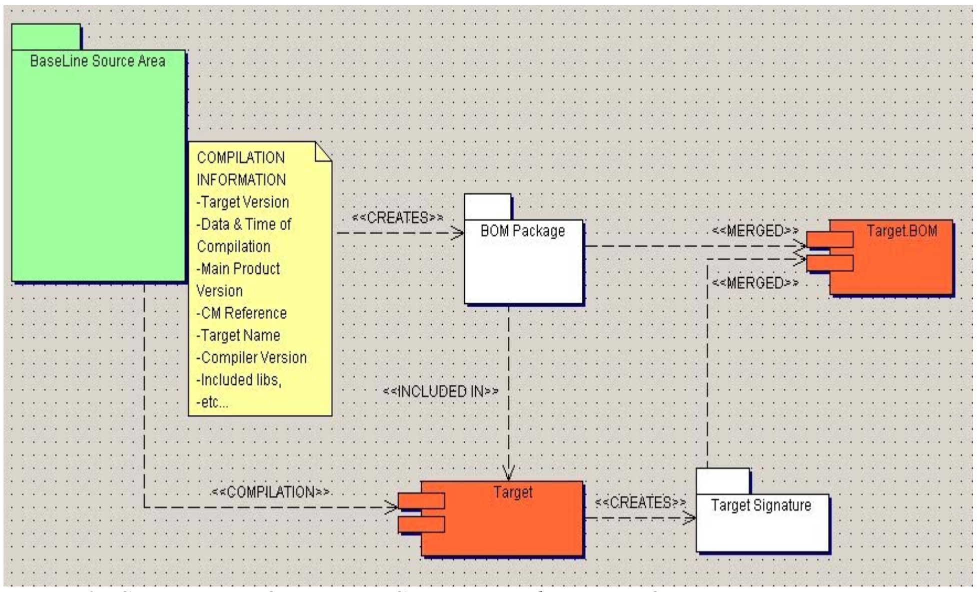


Figure 2: Component BOM, Target Signature and Target BOM

The last step is essential especially in the case where software components (binaries tar or zip archives, etc.) are required by other parties (departments, other companies) for example for product manufacturing. Those manufacturing systems are usually not connected to the CM system and it is therefore essential to have the BOM concept which establishes the link between the otherwise complete decoupled systems.

System BOM Generation

Finally a System BOM (SBOM) is generated out of all component BOMs which are put together for this embedded device. The SBOM contains (despite some general information, like user id, name, date and time) a digital signature which has been calculated out of the digital signatures of all BOMs available in this embedded device.

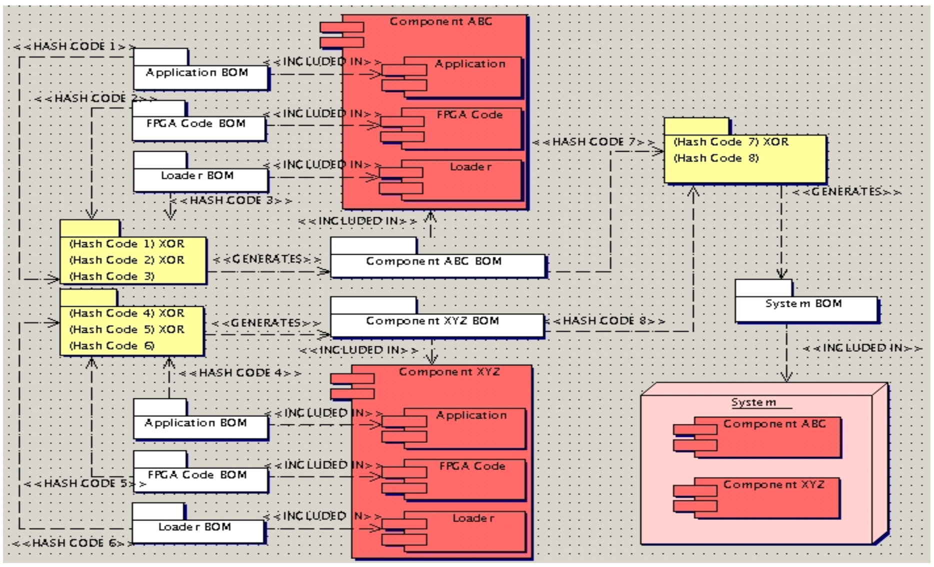


Figure 3: System BOM Generation

The SBOM has to be stored in the embedded device as a component in its own. This opens the path towards device self-validation and self-identification (concerning its version). Now the embedded device would be able to calculate a digital signature for a SBOM from the available BOMs it is composed of and compare this calculated SBOM with an already available SBOM inside. If both SBOMs are equal, then the version and the composition of components have been validated, which results in the fact that this system is running an official version now.

System Self-Validation

In summary this allows a self-validation and identification of an embedded system, the system is able to distinguish its version between:

- official or unofficial version and patches
- identification of official and unofficial ("hacked") components and especially their combinations
- detection of even "just" a re-compilation where no change in the system or component version id has been made.

All this is possible without any access to the company CM system or manufacturing data bases. In addition the embedded device is able to remind a user about the status of its version, without the fact that the user needs to push a "check version button" on a related GUI. This is especially important in those environments where a significant number of embedded devices are tested in very distributed environment, which is the case for example in embedded system development for the oil industry.

Implementation

The BOM has to be included into the build process, however some CM systems do provide already similar concepts and these can be used and integrated into this more generic pattern. However care has to be taken concerning a high coupling with a CM system, this should be avoided as much as possible. Algorithms for the hashcode calculation could be standard 32Bit CRC, MD5 or similar ones. For the calculation of the system BOM hashcode the algorithm could be an XOR operation, a CRC, or again a MD5 algorithm if security is a requirement. Special attention should be paid concerning the version numbering of the component itself, the product and potential patch levels, which have to be part of the BOM as well (encapsulation).
Special attention has to be given to the VHDL based components, as those components are usually available via *.bit* (the output of a VHDL "compiler") files which are generated out of the *.vhd* (VHDL source code) files, by tools (compilers, etc.) provided by several manufactures. However downloading a *.bit* file direct into an embedded device is usually not possible because the file has to be transformed into a *.hex* or similar format. Here is the hook for the build system where additional information (the BOM) can be embedded even within VHDL components. In the future it might be possible to embed a BOM into the *.bit* file direct once supported by the major tool manufacturers.

Known Uses

Q-Marine, a sophisticated and advanced 24bit seismic recording system developed by Schlumberger/WesternGeco for exclusive use by the companies vessels worldwide, which are contracted by oil companies. The system is currently in use in the Golf Of Mexico on a couple of vessels and recently in the North Sea (at the time or writing this pattern). The foundation of this pattern had been laid down somewhere in 2001, the world wide penetration had been finalized almost two years later (with GUI support and the like). However already during the introduction phase (mid 2001) of this pattern, there was not one occurrence anymore, concerning a wrong or a non-certified software version in the recording systems (one recording system contains usually around 100 up to 550 embedded software components). Whereas the time before ......

***C**ontinuus*©, a CM system from *Telelogic*©, now called *CM Synergy*© provides a BOM command "*ccm bom file_spec [file_spec...]*" which is a log of the build commands/results (compiler flags, environmental variable) used to make the finale system, but not for pre-build components. This BOM pattern described here assumes that information about the commands (compiler flags, etc.) are uniquely described by the CM object which is referenced from this here presented BOM. Therefore the log file generated by *CM Synergy*© *BOM* command could be included into this here described BOM pattern.

Known Not Uses

The usage of the BOM pattern might have been beneficial for the recent Mars expedition:

The Mars roboter *Opportunity* was close to a problem at its 20th Sol (a day on Mars). The roboter got the order from earth control to move its "hand" forward, however before this, it should have shifted its ellbow upwards first. The build-in software detected the conflict. "The rover checks commands against potential danger and rejects it if it is too dangerous", said Eric Baumgartner (Jet Propulsion Laboratory). "We did not had the exact same software here on earth and up there on Mars".
Süddeutsche Zeitung, 14. May 2004, Page 10 "Die Marsmenschen von Pasadena"; from Christoph Schrader. Translation by Jürgen Salecker

Benefits

The pattern provides an effective method for device version identification even in the case that the software components (which are composing the embedded device) are still subject to significant change.

The BOM pattern supports a device, or product self-validation concerning its versioning information without the need that a user has to control the version by pushing a "check version button".

The BOM pattern provides a solution concerning individual component versioning, even in the case that different components had to be clustered into one CM object (due to CM performance and constraints given by the required administration).

The BOM provides removable glue between the otherwise complete decoupled build and CM system. It links the build process in time, location, developer, site, patch-level, etc. with the version information provided inside the component and the CM object where the component is hosted.

It supports an agile embedded system development by including cross-cutting concerns for the version identification and calculation and this even over non software components (VHDL, config parameters, etc.) which are a significant part in embedded system developments.

Liabilities

It requires that all components (not only software) which are necessary for a system are accessible in one or more CM systems.

The build rules for even non software components have to be adaptable, so that it is possible to include a BOM generation (a challenge especially with VHDL components).

The software architecture has to provide architectural constructs and interfaces for BOM inclusion and BOM read access. Those methods have to be provided in a generic matter and for all kind of languages, like VHDL, Assembler, Java, C++, C#, etc.

The software architecture has to provide a system BOM component, which serves as a system validation component; it contains the unique key which is the signature for one unique component combination within this system.

Credits

Many thanks to Cyrille Camiul; he did main parts of the implementation of this pattern at WesternGeco/Schlumberger, and greatly helped me with the review. Also many thanks to Arno Haase – my shepherd – who was a great help for me, a novice pattern author.