Enabling System Evolution Through Configuration Management on the Hardware/Software Boundary

René Krikhaar,1, 3 Wim Mosterman,2 Niels Veerman,1, 2, * and Chris Verhoef1

1Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
2Philips Healthcare, Best, The Netherlands
3ICT NoviQ, Eindhoven, The Netherlands

Received 16 May 2008; Revised 21 August 2008; Accepted 24 July 2008, after one or more revisions
Published online 22 January 2009 in Wiley InterScience (www.interscience.wiley.com)
DOI 10.1002/sys.20122

ABSTRACT

As the use of software and electronics in modern products is omnipresent and continuously increasing, companies in the embedded systems industry face increasing complexity in controlling and enabling the evolution of their IT-intensive products. Traditionally, product configurations and their updates were managed separately for the hardware and software discipline. At specified release moments during the development of their products, the hardware and software were released together. But, as the usage, flexibility and complexity of software and electronics increases, and fierce competition requires shorter time-to-market and customizable products, more frequent releases of integrated hardware and software configurations becomes necessary. This evolution requires adequate configuration management both within the hard- and software disciplines and across them. In many organizations, software configuration management is more visibly established than hardware configuration management due to the inherent flexibility and complexity of software. But as the flexibility of hardware has increased through the use of configurable hardware, the need for more intense
hardware configuration management increased as well. To properly enable the evolution of integrated hardware/software systems, adequate configuration management is required in both disciplines. Our article deals with just that: configuration management on the hardware/software boundary, and is mainly focused on the product development phase. The hardware/software boundary has a broad scope; in this article we focus on embedded systems containing software and custom and off-the-shelf electronic hardware, thereby concentrating on configurable hardware containing both software and hardware designs. It is important to note that the concepts of our article apply to configuration management issues that go beyond that scope. We investigated the configuration management practices in six large organizations in the embedded systems industry, including medical devices, defense equipment, optics, document processing, and semiconductor equipment. In particular, we looked at programmable/configurable hardware, as the use and complexity of this technology continue to increase. We propose a generic development cycle with real-life examples to illustrate the configuration management concepts. Then, from the sociotechnical design point of view, we raise awareness and argue that configuration management tools, processes, and its organization must be further aligned to support the complexity and interdisciplinary nature of the hardware/software boundary of evolving embedded systems. © 2009 Wiley Periodicals, Inc. Syst Eng 12: 233–264, 2009

Key words: product configuration management; product data management; software configuration management; hardware/software integration; hardware/software boundary; programmable hardware; configurable hardware; multi-disciplinary configuration management; information management

1. INTRODUCTION

The use of embedded software and electronic components in modern products is omnipresent. Consumer electronics, cars, airplanes and medical devices contain a significant amount of such components, and their use is expected to increase further. Most consumer electronics contain embedded software, and the size and complexity of the software are continuously growing [Graaf, Lormans, and Toetenel, 2003; Ommering et al., 2000]. In the automotive industry, similar trends have been found. For example, the cost of electronics and software can make up 40% of the overall costs of a car [Broy, 2006a; Pretschner, Salzmann, and Stauner, 2004; Thoma, 1999]. Other sources state that 80% of the future automotive innovations will be driven by electronics and 90% by software [Grimm, 2003; Knippel and Schulz, 2004]. These and other complex systems are also called software-intensive systems [Broy, 2006b], and their development and evolution require interdisciplinary teams consisting of experts in hardware, software, and several other disciplines. Such teams must work closely together to meet today's market demands, which are characterized by flexibility, agility, robustness, and adaptability. The automotive industry provides another example, where premium car manufacturers statistically produce no car twice due to a high number of variants imposed [Fricke and Schulz, 2005]. A free market requires both competitive pricing and flexibility, which is often implemented by balancing between dedicated hardware and software on standard components. Therefore, evolving software in products will increase in importance in many industries.

Simultaneously, the boundaries between the hard- and software disciplines are vanishing. The increasing combination of software with hardware requires tight cooperation between the disciplines and early integration, in particular during the development phase of a product's lifecycle. Hard- and software components share more and more dependencies, up to the level of components consisting of both software and hardware [Adams and Thomas, 1996]. Furthermore, decisions to implement functionality in software, hardware, or both, are more common now, including the fact that these decisions change during a product's lifecycle [Adams and Thomas, 1996; Dahlqvist, Crnkovic, and Asklund, 2004]. For instance, during the design phase of automotive ECUs—Electronic Control Units—software is tested in a Hardware-in-the-Loop system to simulate driver, vehicle and vehicle environment [Stoermer and Verhoef, 2008]. Hardware-in-the-Loop provides early feedback on design decisions and early detection of design mistakes. In a later stage, the simulated hardware is physically implemented and integrated with the software, and used for real-time tests.
The so-called partitioning decisions require specific techniques for design partitioning and hardware/software codesign [Ernst, 1998; Wolf, 2003]. But the vanishing boundaries between hardware and software also pose several requirements for the development processes and the supporting tools. As the software and hardware development disciplines have different origins and requirements, the alignment of these disciplines has various challenges, such as process adjustments, information exchange, tool integration, and cultural differences [Dahlqvist, Crnkovic, and Asklund, 2004]. The field of configuration management is instrumental to support many of these challenges.

Configuration management is deployed in situations where various objects can be combined in various ways to form valid configurations. The term is often associated with service management of product configurations where various objects can be combined in various ways to form valid configurations. For completeness’ sake, we review some definitions here. A definition from the early 1970s states [Samaras and Czerwinski, 1971, page ix]:

Configuration management is the discipline of ensuring that equipment or hardware meets carefully defined functional, mechanical, and electrical requirements and that any changes in these requirements are rigidly controlled, carefully identified, and accurately recorded.

A more recent definition from the International Organization for Standardization (ISO) states [ISO, 2003, page v]:

Configuration management is a management activity that applies technical and administrative direction over the lifecycle of a product, its configuration items, and related product configuration information.

The Electronic Industries Alliance standard (EIA) for configuration management, which is used in many industries, states [Government Electronics and Information Technology Association, 2004, page v]:

Configuration management applies appropriate processes and tools to establish and maintain consistency between the product and the product requirements and attributes defined in product configuration information.

The above definitions show that configuration management (CM for short) is a traditional discipline. CM as a concept was already known in the early 1900s to organize the production of automobiles [Bazelmans, 1985]. Despite the early use of the CM concept, armored ground vehicle systems rolling-off the production line were still inconsistent during World War II, resulting in multiple configurations that went undiscovered until maintenance [Newborn, 1997]. Those uncontrolled changes in the weapons industry led to the formal establishment of CM in the 1950s [Samaras and Czerwinski, 1971]. However, the traditional view on CM as a control function has given CM a bureaucratic taste for many. But, in fact, CM is an enabler for system evolution: It is an essential supporting function for project management and product assurance [Bersoff, 1984; Kidd, 2001]. Configuration management, with its change control boards reporting directly to the program management provides the control necessary for all the teams working on a complex development to be “marching to the same drum”—that is, the latest approved baseline. Indeed, a recent benchmark report [Aberdeen Group, 2007] mentions the following as the main pressures driving CM improvements: product quality improvement, time-to-market improvement, and development cost reduction. By supporting efficient building and testing of correct configurations, by planning and tracking configuration baselines, by supporting efficient problem solving, by avoiding unnecessary rework, and by controlling changes to ensure quality levels, CM is a true enabler for system evolution.

In addition to the above benefits of CM, many organizations are subject to external requirements that stress the importance of CM. For instance, medical device manufacturers that sell on the U.S. market must comply with the U.S. Food and Drug Administration regulations for design control [U.S. FDA, 1997]. These regulations state that medical device manufacturers must be able to demonstrate that the design of a device was developed in accordance with the approved design plan and requirements. A manufacturer is accountable for this requirement for 15 years after initial shipment of a device. This requirement implies that the product defining data and product design data must be archived properly such that long-term traceability is guaranteed. The design data must be archived properly such that long-term traceability is guaranteed. The design data include (but are not limited to) the source code for a device. Hence, sources must be stored properly and remain fully accessible for 15 years onwards. If a manufacturer fails to comply, the consequences can have a severe impact. In the regulations document [U.S. FDA, 1997: 43], a case is described where a manufacturer did
not posses the software source code of a part that behaved erratically in the field. The component supplier had withdrawn without granting the manufacturer the right to access the source code. Therefore, there was no practical way to maintain the software. In the end, the manufacturer was forced to recall the product, and all known units were collected and destroyed. For more real-life examples, we refer the reader to the website of the U.S. Food and Drug Administration [U.S. FDA, 1997]. In the “Warning Letters” section, a query on “design control” reveals design control issues found at several medical device manufacturers.

The above example illustrates the realities of CM. Adequate archiving, controlling, and traceability of product defining and product design data, including the software source code and hardware design sources, is simply business-critical to many organizations. The link between product design data and product defining data is first established in the development phase, when both types of data are being created. Hence, the example emphasizes the integral nature of CM: It is essential during the entire lifecycle of a product, starting at the first base-lined requirements of a product and ending with its retirement. All aspects of a product, including development, commercial, production, and service aspects, are in fact subject to and supported by CM.

In our article, we will focus on the product development phase, more specifically, from requirements until production start-up. During that phase, the foundations for CM and its subsequent lifecycle phases are put in place. Where important, we will also discuss production and service aspects of CM.

**Hardware and software configuration management.** The term *hardware* is widely used in different ways. Some domains refer to hardware to indicate mechanical structures, factory equipment, entire transportation systems, and more. In engineering, hardware usually refers to mechanical and electrical objects. But the classical view of software engineering people on hardware is limited to the computer equipment on which their software runs. Our article defines hardware as the physical components of a product, including mechanical and electrical parts. These components range from custom designed and developed in-house or by a supplier, to bought off-the-shelf. We limit the scope to electrical parts, but it is important to note that the concepts of our article apply to CM issues that go beyond that scope.

Traditionally, CM has been geared towards controlling product data during product development, in particular for data on hardware objects. The focus of classical hardware CM is the control of documentation such as drawings. Control is exercised through identification numbers, the concept of interchangeability, structuring bills of materials, procedures for product and document release, and change control processes [Watts, 2000].

A derived discipline of CM is *Software Configuration Management* (SCM), which emerged in the 1980s. SCM was a response to the many failures in the software industry throughout the 1970s, when SCM was considered a less mature discipline than hardware configuration management (or HCM) [Babich, 1986; Bersoff, 1984]. Soon, CM for software was regarded as an important cornerstone for managing software projects [Buckle, 1984]. Later on, the CM function configuration control was considered as a key risk issue in software intensive projects [Conrow and Shishido, 1997]. In Bernstein and Yuhas [2005], CM is defined as a systematic process that controls and tracks all changes to a software product over its lifetime, and considered an inline function instead of a support function. SCM ultimately supports answering the “what’s changed” questions in order to support efficient problem solving [Bernstein and Yuhas, 2005: 393–394]: what changes were requested, and by whom? What changes have been accepted in the current build? Which release contains what changes? Which components were modified? Who is responsible for a change? And so on. With the advent of software maturity models, such as CMM(I) [Carnegie Mellon SEI, 2006], SCM is widely acknowledged as an essential discipline for product development. In fact, some people consider SCM in general to be more mature than HCM, but this is probably caused by the attention that SCM received during the past 20 years.1

SCM differs from HCM, in that it must support the distinct aspects of the development and production processes for software. For example, software development iterations are generally more frequent, the building of software executables is often fully automated on a regular, sometimes daily basis, and reproduction of software executables is practically without cost. Concurrent development of software objects through branching and merging of development lines is common practice. However, the malleability of software provides both opportunities and threats. Although branched development increases speed-to-market, it requires proper modeling to manage its vast complexity [Berczuk and Appleton, 2003; Walrad and Strom, 2002]. Moreover, the flexibility of software allows for last-minute changes that can result in bad fixes and patches [Jones, 2007: 579–580].

---

1One of the anonymous reviewers pointed out that the world of mechanical and electrical design had mature CM processes before the term “software engineering” was even coined.
Hardware is not as malleable as software; a hardware development iteration takes generally longer than a software iteration, as the production of physical prototypes is usually long and costly [Dahlqvist, Crnkovic, and Asklund, 2004]. In addition, changes to manufactured hardware objects require their physical modification. To perform tests, it is usually necessary to alter a hardware test configuration, which takes generally more time than for software. These aspects must all be calculated in advance to allow for proper planning on scarce testing resources, but also the correct configuration information is required in order to perform valid tests. With the advent of CAD systems for mechanical and electronic design, nearly all engineering data is now stored in softcopy form and became as changeable as software. Several types of changes to designed hardware are now verifiable electronically before actual manufacturing of a physical part. The use of Hardware-in-the-Loop further alleviates the limitations and CM issues of the physical hardware world. However, as soon as the first hardware prototypes have been manufactured, the appropriate CM processes for hardware must be applied as well, as the impact on logistics and stock must be considered when changes are proposed.

Configuration management tools. As the lifecycle processes of software and hardware are different, so are the accompanying CM processes and practices [Dahlqvist, Crnkovic, and Asklund, 2004]. CM heavily relies on tools; hence, the current tools’ functionality reflects the distinct practices for software and hardware. Software CM is supported by SCM tools, providing functions such as version management [Conradi and Westfechtel, 1998; Tichy, 1985], workspace management [Estublier et al., 2005], build management [Sjoberg et al., 1997], change and defect management [Bays, 1999], release management [Bays, 1999; Hoek and Wolf, 2003], and distributed and concurrent development [Blackburn, Scudder, and Van Wassenhove, 2000]. SCM tools have existed since the 1970s, when tools such as make [Feldman, 1979] and SCCS [Rochkind, 1975] were developed. The tool make is the counterpart of the assembly process in hardware; it defines the steps required for assembling the product. Software building (or manufacturing [Bernstein and Yuhas, 1978]) can be performed very efficiently or inefficiently; objects that are used several times should be built only once. There are exceptions especially in the case of multisite deployment of complex software products with location-specific aspects, where build-costs are a fraction of installation and deployment costs [Faust and Verhoeft, 2003]. Software building even spawned a separate SCM function: build management [Berczuk and Appleton, 2002; Maraia, 2005]. SCM tools are essential to support modern SCM and software development. The software community is working to improve integration of these tools in approaches, such as task-oriented configuration control and Unified Change Management [IBM, 2008c].

Hardware (and product) CM is generally supported by Product Data Management (PDM) functionality [Philpotts, 1996]. PDM concerns the discipline of designing and controlling the evolution of a product design [Crnkovic, Dahlqvist, and Svensson, 2001]. PDM systems are used to support the management of any data that is required to design and manufacture products, such as drawings and production routings, design data, project plans, and product structures and configurations. The first commercially available PDM systems were introduced in the early 1980s [Philpotts, 1996]. As PDM systems are used to manage any information needed throughout a product’s life, PDM forms the backbone for HCM in many organizations.

Configuration management research. Intensive research on software (and system) CM by industry and academia has continued until the beginning of the 21st century [Dahlqvist and Whitehead, 2005]. An elaborate overview of SCM research achievements is given in [Estublier et al., 2005]. Differences and similarities for SCM and PDM are reported in Westfechtel and Conradi 1998]. Although SCM and PDM have similar goals, PDM emphasizes modeling and SCM emphasizes building, evolution, and concurrent engineering [Estublier and Vega, 2007]. Dart [1992] already identified several similarities in supporting electronic hardware design and software development. It was observed that both the software and hardware communities are addressing the same problems, and that a unified CM approach for software and hardware would be beneficial. Although there are attempts to unify the disciplines, it is clear that the disciplines have evolved mainly in isolation.

An important research direction is the integration of the hardware (mechanics and electronics) and software development processes from the information management viewpoint [Svensson and Crnkovic, 2002]. This approach requires integration of PDM and SCM systems [Crnkovic, Asklund, and Dahlqvist, 2003; Estublier, 2000; Estublier, Favre, and Marat, 1998] but also the processes, as the border between the software and hardware disciplines is vanishing. Effort is being put into integrating software and hardware development processes and the accompanying tools. There are various challenges that complicate these integrations, which are well explained in [Dahlqvist, Crnkovic, and Asklund, 2004]: process adjustment; information exchange, access, and flow; infrastructure support; tool integration; cultural differences; and so on. Research on SCM and PDM integration is being performed from
both theory and practice viewpoints, and the challenges are approached by proposing full integration (e.g., unified models for PDM and SCM [El-khoury, 2005; Lee, 1997; Nguyen, 2006]), loose integration, and no integration between SCM and PDM implementations [Crnkovic and Park, 2003]. Many challenges come from integrating existing SCM and PDM systems, as these systems have not been designed with integration in mind.

The described challenges are in line with the concepts of socio-technical design theory, which emphasizes the balancing of technical and human factors in design processes [Mumford, 2006]. We believe that the main focus for the hardware/software boundary must be on process and technology alignment, because it can deliver significant benefit for the design process.

The hardware/software boundary. A technology that is precisely on the boundary of hardware and software is configurable hardware [Garcia et al., 2006]. Configurable hardware devices are also known as programmable logic [Pellerin and Holley, 1991], programmable hardware [Magione-Smith et al., 1997], evolvable hardware [Greenwood and Holley, 2006], and complex electronics [Berens, 2004]. Configurable hardware originates from the early 1970s, and concerns electronics devices whose behavior can be (re-)defined after initial production, by modifying their internal structure. These devices are basically customizable integrated circuits, and are often designed with hardware description languages (HDLs) such as Verilog [IEEE, 2005a] and VHDL [IEEE, 2002] (Very high speed integrated circuit Hardware Description Language). Although there exist languages for hardware design that were clearly inspired by “software languages,” such as SystemC [IEEE, 2005b], these are not as widely used for hardware as the hardware description languages VHDL and Verilog. Some argue that this is caused by the specific requirements of hardware languages [Edwards, 2005]. Configurable hardware provides increased flexibility compared to fixed hardware and is used to achieve significant performance benefits compared to conventional computing in many application areas [DeHon and Wawrzynek, 1999; Garcia et al., 2006; Magione-Smith et al., 1997]. Configurable hardware devices, such as the CPLD (Complex Programmable Logic Device) and FPGA (Field Programmable Gate Array), originate from the hardware world, but have software-like flexibility, which allows for a shorter time-to-market and flexible customization after initial manufacturing. Another area where flexible reconfiguration provides value is factory automation. Since factory machines are often redesigned, replaced, and upgraded, the advantages of reconfigurability in automation computers and software provide major advances in flexibility and speed in manufacturing.

For illustration purposes, we zoom in on an electronics architecture. Figure 1 shows a hardware architecture containing configurable hardware; we adopted the figure from [Jaring, 2005]. The hardware components are standard parts in a fixed assembly, but the functionality of the configurable hardware device is (re-)configurable. We summarize the description of the architecture here.

- FPGA: a configurable hardware device, containing standard and custom logic. For instance, signal or image processing logic.
- Microcontroller: a microprocessor with additional peripherals in a single chip; it is used for coordination tasks such as initializing the configurable hardware, but also for running an (embedded) operating system.
- Data memory: location for storing volatile microcontroller and configurable hardware data during execution.
- Program memory: location for storing the microcontroller program and the precompiled configurable hardware configuration load modules.
- Memory controller: bus connections for data and program memory.
- Configuration controller: bus connection and configuration protocol implementation for reconfiguring the configurable hardware.
- PCI controller: Input/Output bus for interfacing with the rest of the system, i.e. other hardware and the main system software.

Although most of the described components are usually found in electronic systems, the microcontroller and configurable hardware often form the center of complex electronic systems. Typical functionality in

![Figure 1. An electronics architecture containing configurable hardware [Jaring, 2005].](image-url)
electronic systems ranges from simple bootloader code to dedicated signal processing algorithms and operating systems. Functionality is either implemented in software or in configurable hardware technology, as the two technologies complement or replace each other. In fact, both technologies are even implemented in a single hardware device using an embedded processor; this is called embedded processing. Due to its versatility and flexibility, the use of configurable hardware in electronic systems has increased over the years, as it combines many advantages of both hard- and software. Below we will mention the most prominent ones, originating from sources reporting on the automotive industry.

- **Configurability** of hardware offers the potential for flexible field upgrades and fixes, and the ability to create multiple product variants with a single piece of hardware [Morris, 2007]. Configurable hardware allows integration of fixed hardware components into a single hardware component, thereby reducing the size of a hardware design and also increasing its flexibility.

- **FPGA and ASICs.** FPGAs provide increased flexibility and reuse potential than Application Specific Integrated Circuits, which suffer from high nonrecurring engineering costs, long design cycles, increased time-to-market, reduced flexibility, and higher risk [Mason, 2006].

- **FPGA and MCUs.** Microcontrollers, or MCUs, provide flexibility and consume relatively large amounts of power. Furthermore, MCUs cannot satisfy the extreme response-time requirements of automotive applications such as control units for state-of-the-art diesel engines. The closed-loop control units must be able to monitor and adjust fuel injection on a cycle-by-cycle basis in real-time [Maxfield, 2007]. In contrast with MCUs, FPGAs offer designers better performance and features such as I/Os, flexible logic resources, and more [Mason, 2006].

But, of course, there are also disadvantages regarding configurable hardware. We mention the most important ones, being reported by various sources:

- **Complexity.** The adoption of FPGAs in some application areas is currently limited by the complexity of FPGA design. Compared to conventional software, FPGA designs have long compilation cycles, even after minor changes to the source code [Wikipedia, 2007].

- **Security.** The manufacturer’s intellectual property, in the form of the FPGA or processor operating code and logic, typically is unprotected and can be accessed for reverse engineering, duplication or unauthorized modifications [Howell, 2007].

- **Power usage.** FPGAs are not commonly used in battery-powered applications because they consume more power than ASICs and lack power management features [Tuan et al., 2006].

- **Operating temperature.** The relatively high power consumption associated with FPGAs can cause self-heating, which limits the larger devices to a maximum operating temperature of around 100°C. This makes them unsuitable for operation in high temperature environments, such as under the hood of a car. FPGAs are also susceptible to firm errors, i.e., loss of the configuration data. Innovations in this respect seem apparent; for instance, hardware manufacturer Actel claims to have FPGAs that meet the requirements of automotive ICs: being able to operate in high temperature environments and immune to firm errors [Maxfield, 2007].

Due to the increased use of configurable hardware, many organizations face increased complexity in its development. The sizes of the hardware designs, as well as the number of developers on a single design, have gradually increased, posing challenges for CM of mixed hardware and software configurations. However, the established CM practices for configurable hardware originate from the hardware world and have often evolved on an as-needed basis. Hence, the CM processes and accompanying tool support are usually not able to adequately support development processes on the hardware/software boundary.

**Related work.** Work that seems strongly related at first sight is about optimizing dynamic configuration of the content of configurable hardware devices. For example, Zhiyuan [2002] examines CM techniques for reconfigurable computing, which deals with run-time reconfiguration of configurable hardware devices. The goal of that work is to improve the reconfiguration time and resource usage of a device. In Garcia et al. [2006: 9], configuration overhead management techniques for configurable hardware are discussed. In our article we deal with system level CM of software, electronic hardware, and configurable hardware to improve development time and product quality.

Stoica and Andrei [2007] discuss the increased flexibility in adaptive and evolvable hardware. Evolvable hardware is about self-adaptive systems using configurable hardware; the formal aspects of adaptive behavior are described by the founders of cybernetics [e.g., Wiener, 1965]. The authors define a configurable hardware classification, and identify several types of adap-
tations to configurable hardware and link those to the different lifecycle phases of a product. Becker and Hartenstein [2003] also attempt to clarify the confusing terminology surrounding configurable computing. They introduce the terms “morphware” and “configware.” The former refers to the physical hardware device that can be configured, and the latter to the hardware configuration load module that is loaded into the hardware device and instantiates its behavior. Although the authors propose new terms to avoid confusion, we deem it unlikely that the terms will be adopted widely, due to the widespread existing terminology. In our article we therefore stick to the terms fixed hardware and configurable hardware.

The work by Buckley [1995] is a comprehensive resource on CM. In Buckley’s book, the term firmware is defined as a hardware part that has a software load module added to it, but it is also mentioned that the term is confusing. There is no classification of software load modules and hardware load modules, as from a technical and CM viewpoint these can be handled in a similar way. Buckley acknowledges that CM of hardware in which software is embedded poses additional challenges for CM, especially when the hardware is remotely reprogrammable/reconfigurable [Buckley, 1995: 114–116]. Kääriäinen, Taramaa, and Alenius [2004] describe experiences with CM support for embedded systems. They present several observations on the planning and role of CM, the use of CM tools, and CM of common interfaces, for example, the SW-SW and SW-HW interface.

Ronkainen, Taramaa, and Savujoa [2002] focus on process improvement of hardware-related software in the telecommunications industry, and distinguish between standard and custom hardware and software components. Standard hardware is usually off-the-shelf commercially available; think of microprocessors or microcontrollers. Custom hardware is tailored to specific needs: an application-specific integrated circuit. The focus of their article is mainly on digital signal processing software development, which is tightly coupled with hardware. Most of the functionality in telecommunications devices used to be done in hardware, but this is tilting towards software. The authors state that the traditional software development methods tend to fall short on the hardware/software boundary. They also mention that in hardware-related software development, real-time requirements fundamentally influence architecture design, and those requirements manifest themselves also in the coding phase. The required iterative nature of interdisciplinary development depends on timely and accurate information exchange between hardware and software developers; hence, issues arise from the different backgrounds and vocabulary of hardware and software designers.

Auer and Taramaa [1996] describe improvement levels for software CM of embedded software. The authors sketch steps from version control to configuration and product management, and present an incremental approach to develop CM maturity. Their article focuses on software CM, but the authors acknowledge the challenges for integration with hardware components. Conradi and Westfechtel [1999] identified common CM on hybrid products consisting of electrical, mechanical, and software components as a research theme as well. Our work addresses the issues identified by others, by studying CM on the border of hardware and software.

Context and outline. This work is about hardware and software development and CM from a systems development perspective. In particular, we focus on configurable hardware, which is on the hardware/software boundary. At that point, the hardware and software worlds come together. We recognize that configurable hardware devices share the late binding inherent in software design, and that this late binding demands different approaches to CM. From the socio-technical design point of view, we argue that CM tools and processes must be employed and aligned to properly support the interdisciplinary nature of the boundary and to control and enable system evolution. We also provide concrete examples why the hardware and software world must be aligned as such.

The work in this research was done in several steps. We started with a short survey to map the configurable hardware/software CM situation in six organizations using questionnaires, showing that it is still somewhat manual in comparison with software CM. Then, in a follow-up meeting, we clarified the responses. After that, we investigated the situation at three of the organizations in more detail. Based on the results of the questionnaires and our findings and experience, we propose a generic development cycle for commercial product development including both software and hardware, as a means to highlight the CM issues. We discuss important CM issues that can cause breakdown in product development and delivery, that derive from the rudimentary nature of CM for configurable devices. Subsequently, we discuss the steps to take to align the hardware and software world, to support the challenges on the hardware/software boundary.

The remainder of our article is divided into four sections. In Section 2, we discuss the hardware/software boundary. We propose a clarification on the boundary with regard to software, hardware, and configurable hardware. In addition, we present the results of the CM survey among the six organizations. In
Section 3, we describe a generic development cycle for electronics. We explain the details of development and CM of configurable hardware, in the context of software, hardware, and system development. In Section 4, we raise awareness on the issues that arise when the hardware and software worlds must work together, in particular on the hardware/software boundary. Finally, in Section 5 we conclude. Due to the number of acronyms in this article, a glossary is compiled at the end.

2. THE HARDWARE/SOFTWARE BOUNDARY

In this section, we clarify the hardware/software boundary and model the functional flexibility of the different technologies. After that, we discuss the most prominent differences and similarities between software and configurable hardware development. Finally, we present the results of a survey of configurable hardware in six companies to illuminate the state-of-the-practice.

Hardware and software are terms existing for many years now. Their production means dramatically changed during these years. Producing software evolved from the era of the punch card, to writing assembly code, to writing programs in a programming language (e.g., PL/I, Cobol), to more abstract languages (object orientation), to drawing diagrams from which code is generated. A similar situation holds for hardware. In the beginning it was a voluminous work using cables, relays, and soldering irons. Breadboards simplified this world due to the miniaturization of the hardware components used for building the hardware system. This approach evolved into dedicated chips with nand and nor ports, into more sophisticated CAD environments that support design of chip layouts for producing dedicated products such as ASICs. Hardware Description languages (HDLs) emerged as major vehicles to describe functionality for hardware, and enabled simulating the description on a general purpose processor. Nowadays, models and hardware are being generated for configurable devices, such as FPGAs, up to complete systems in a single device: the System-on-Chip (SoC). For SoC designs, dedicated languages are created to program the chip. In fact, it is a microprocessor including some extra special-purpose hardware, for example, video processing. For this hardware special languages plus the accompanying compilers are defined, which are often based on languages like C. Both hard- and software descriptions became more abstracted from the physical device and are now more flexible in modifying the description itself, but also to change the physical device.

Terminology for hardware and for software has been established merely in isolation, but the moving hardware/software boundary confuses existing definitions. For the purpose of our article, we categorized some software and hardware terms.

2.1. Terminology on the HW/SW Boundary

The hardware/software boundary is surrounded by several terms. Here, we explain the most prominent related to configurable hardware that we will use in our article. Where appropriate, we adhere to the IEEE Standard Glossary of Software Engineering Terminology (IEEE Standard 610.12-1990 [IEEE, 1990]). To illustrate the terms, we use a generic electronics architecture. Figure 2 shows the architecture with several hardware/software terms. We explain each term below.

**Software** appears in many forms: boot code, operating systems, databases, compilers, end-user applications, and so on. Software may include the software documentation, software source files, and software executables, and is usually developed by software developers. The IEEE [1990] definition for software is:

Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Software that is part of a larger (embedded) system is called embedded software. The IEEE definition for embedded software is:

Software that is part of a larger system and performs some of the requirements of that system; for example, software used in an aircraft or rapid transit system.

Much software is part of a larger whole; hence, a lot of software qualifies as embedded software. In Figure 2, two different software systems are distinguished: embedded software in the electronics architecture and the main system software. The former is closely coupled with the hardware, while the latter can provide updates to the embedded software and to the configurable hardware in the electronics architecture. Note that main system software can be embedded software as well according the IEEE definition. Hence, embedded software is a term surrounded by confusion, but we will not attempt to redefine it here. We avoid its usage in our article, and use the term main system software frequently to indicate a “master” system that initiates, controls, and updates the software and subsystems in electronics.

To make a clear distinction between software and configurable hardware, software concerns machine instructions that are executed by a microprocessor with a
Figure 2. The hardware/software boundary, illustrated by a simple electronics architecture.
fixed instruction set. We discuss this distinction in more detail later. When necessary, we will mention whether the term software refers to documentation, source files, or executables.

**Firmware** is not shown in Figure 2, but we briefly discuss it for completeness’ sake. Firmware is another term that is causing confusion, which can also be read in the IEEE definition for firmware:

The combination of a hardware device and computer instructions and data that reside as read-only software on that device. Notes: (1) This term is sometimes used to refer only to the hardware device or only to the computer instructions or data, but these meanings are deprecated. (2) The confusion surrounding this term has led to suggest that it be avoided altogether.

Firmware is used to refer to software load modules in on-board, nonvolatile memory devices, such as a PROM (Programmable Read-Only Memory), or in peripheral memory devices, such as hard drives. On-board memory devices used to be read-only, but have evolved and can be reprogrammed. Examples are the EPROM, short for Erasable PROM, or flash memory. As such devices are no longer read-only, the definition of firmware seems to be outdated as well. Also, as load modules can be stored on hard drives and from there loaded into the on-board memory during run-time by the main system software, the confusion increases: When the load module is on the hard drive, it is considered to be software, but as soon as it is stored in a PROM, it turns into firmware. Nevertheless, the term is widely in use, and it is used for hardware load modules as well. We will not use the term firmware in our article to avoid confusion.

**Fixed hardware** concerns physical objects consisting of mechanics and electronics. The software view on hardware is usually limited to computer equipment; this view is illustrated by the IEEE definition for hardware:

Physical equipment used to process, store, or transmit computer programs or data.

This definition of hardware is formulated from a computer systems viewpoint and does not encompass mechanical hardware objects, such as nuts and bolts, or mechatronics for that matter. We use the term fixed hardware for hardware components and assemblies, to indicate its flexibility with regard to changing its functionality. Fixed hardware includes microprocessors and controllers, memory, converters, resistors, and so on, but also mechanical hardware. Although some of these examples are considered programmable, for instance, memory or a processor, their functionality is fixed. A processor has a fixed instruction set, and a memory device can only be used to store and retrieve data. Often, fixed hardware components are commercial-off-the-shelf components. The individual components are assembled on an electronic printed circuit board, usually designed in-house but mass produced by an external manufacturer. Hardware allows for fast execution, but this is traded off against flexibility, as the design and production time of a printed circuit board easily take several months.

An example of a complex fixed hardware device is the Application Specific Integrated Circuit, or ASIC, which is an integrated circuit customized for a particular use. Compared to FPGAs, ASICs are usually smaller, use less power, and execute faster, but the nonrecurring engineering cost for an ASIC is high, which makes them suited for high volume production (in fact, a microprocessor is an ASIC). ASICs are often designed using HDLs and validated using FPGAs [Turley, 2003], but in general ASICs cannot be modified after production and therefore we consider them to be fixed hardware.

**Configurable hardware** consists of a design and a device. Examples of such devices are an FPGA (Field Programmable Gate Array), a CPLD (Complex Programmable Logic Device) or PAL (Programmable Array Logic). Configurable hardware is generally designed with HDLs and can be reconfigured after production of the device. The act of configuring a device involves loading a hardware configuration load module into the logic structure of the device. Note that the generated hardware design file is also called an image file, load module, or configuration bitstream. After the configuration module has been loaded, the device is instantiated with the designed behavior and ready for operation. Configurable hardware devices can even be reconfigured during run-time, which is referred to as run-time reconfiguration or RTR [Compton and Hauck, 2002]. RTR offers the possibility to modify part of the content of a device during run-time, creating even more flexibility, at the cost of increased configuration complexity.

Configurable hardware is usually designed by hardware designers, unless the device contains software running on an embedded microprocessor. Then the device is a joint effort of both disciplines, and is called a System-on-Chip. A System-on-Chip (SoC) [Voros and Masselos, 2005] design establishes a more tight integration of hardware and software. An SoC design is an integration of several components of an electronics system into a single chip, for example, an ASIC or FPGA, or a combination of these. It usually contains an embedded microprocessor with additional peripheral devices and software. The integration of the compo-
ments has several advantages, such as reduced power consumption, reduced assembly costs, and increased flexibility when implemented using configurable hardware. However, the costs of an integrated custom chip are often higher than when using several standard components. The increasing use of standard off-the-shelf components for configurable hardware, or so-called IP cores, may change this. SoCs have brought the hardware and software disciplines even closer to each other, as the results of both disciplines are integrated into a single chip. If an SoC is implemented using an ASIC, its functionality is fixed and hence it is fixed hardware. Note that an SoC implemented using an FPGA can even be configured at runtime.

We use the term configurable hardware to refer to hardware whose behavior can be reconfigured after production by a hardware design. When we speak of configurable hardware development, we refer to the development of the hardware design, not the development of the (potentially un-instantiated) hardware device itself, as this is done by the configurable hardware device manufacturer.

Summary. Terminology surrounding the hardware/software boundary is not unambiguously defined. This is also caused by the fact that the hardware/software boundary is not clearly demarcated, and the implementation of functionality is not restricted to either hardware or software but crosses the borders. The decision to implement functionality in either hardware or software is subject to requirements such as flexibility, speed, power consumption, and costs.

2.2. Software Engineering versus Hardware Engineering

Configurable hardware is on the hardware/software boundary, as it shares several properties with both the hardware and software disciplines. Here, we discuss the similarities and differences that are present in hardware, software, and configurable hardware.

In Figure 3 we propose a flexibility model of fixed hardware, configurable hardware, and software, based on an existing software variability model from [Jaring, Krikhaar, and Bosch, 2004]. The figure illustrates the flexibility of hardware and software at the various binding phases. In each binding phase, configuration decisions are made that affect the downstream phases. Configuration decisions that are made at a later binding stage indicate more flexibility in functionality. Configurable hardware has a later binding stage than fixed hardware. As a configurable hardware design is synthesized and loaded into a device, it will exhibit the modeled behavior. In contrast to fixed hardware, a device is

![Figure 3. Flexibility of fixed hardware, configurable hardware, and software.](image-url)
Software programs and hardware designs. A design for a configurable hardware device is often modeled using an HDL that describes its behavior (e.g., VHDL [IEEE, 2002] or Verilog [IEEE, 2005a]; see Smith [1996] for a comparison of these languages). An HDL describes the circuit’s operation, its design and organization, and tests to simulate and verify its operation. HDLs and software languages coincide regarding the language constructs. For instance, the VHDL language has been based on the Ada language to avoid reinventing concepts that were proven in Ada. Hence, it is not surprising that some people think of hardware descriptions as “just software.” Hardware descriptions are flexible, often text-based: Late changes are possible, and the hardware has strong potential for reuse. But there is a significant difference with regular software programs. HDLs have strong capabilities for the description of a concurrent system, that is, several processes that are executed concurrently. For example, the result of two VHDL assignment statements that appear sequentially in a program can be produced at exactly the same time. In a software language, instructions are ultimately executed sequentially (unless several processors are employed, but this requires proper coordination). From a developer’s point of view, developing configurable hardware requires strong parallel programming skills and proficiency in hardware constraints, such as timing and memory. On the contrary, for many software developers, hardware constraints do no longer play a prominent role, as memory for application software is not as limited and expensive as it used to be. There are several other differences between hardware and software languages; for details we refer the reader to [Duckworth, 2005].

So, what ultimately resides in a configurable hardware device is not a software program. The main difference is that a software program specifies computer instructions, whereas a hardware design specifies a logic structure. This logic structure is basically the instruction set for the configurable hardware device. An HDL is used to define the structure [Beren, 2004; Duckworth, 2005], and the actual inputs to a device during operation—the instructions—are (concurrent) signals that must be processed.

Graphical and textual design. Hardware development originates from graphical design using graphical schematics from which the hardware description is eventually generated and synthesized. Although the software community is currently eager to employ graphical techniques to develop software (e.g., model-driven engineering), we believe that the majority of software programs are programmed and maintained at the text level; hence, think of the large amount of legacy software that is currently being maintained in many organizations. Graphical design may be more suitable for humans, but text-based objects are more suitable for processing by tools. For instance, to compute differences between two objects, textual differences are more convenient to compute than graphical ones. The size of a hardware design for a single device can be in the range of tens of thousand of lines of code, plus graphical designs. Hence, several developers must work together on a single design, and reuse of (parts of) designs is attractive. Similar trends were observed in software development in the 1980s, and supporting branch and merge practices for software engineering have been developed.

Configuration management. From a CM point of view, hardware description source code can generally be treated in the same way as software source code, and configurable hardware load modules can be treated in the same way as software load modules for an electronics device, for example, embedded software such as microcontroller code. The release of both types of load modules can require releases towards the SCM and PDM systems, as there are dependencies with both systems. The load modules are required during production of electronic hardware boards to perform basic tests, but may also be needed for upgrades after production through main system software updates. Hence, both development processes share these complexity properties, and during integration steps baselines of both disciplines and their interfaces must be established. As these are all subject to changes, they must be controlled properly. There are several versions of hardware and software objects, and some versions are compatible while others are not. Therefore, configuration baselines must be established that cross the disciplines’ boundaries.

Summary. Configurable hardware illustrates the vanishing boundary between the hardware and software worlds. The offered flexibility of integrated hardware and software has posed new challenges for the involved disciplines, as several differences and similarities have been observed. In particular, configuration management challenges have appeared, as systems of mixed hardware and software configurations are being developed, released, and maintained.
2.3. A Small Survey on Configurable Hardware

To illuminate the state-of-the-practice on configurable hardware, we conducted a small survey among six large organizations. We identified the organizations with names Organization A through Organization F. All of the organizations manufacture products with a significant amount of software and electronics; their products contain several millions of lines of software source code. The six organizations are operating in the following industry sectors: medical devices (two organizations), semiconductor equipment, optics, document processing, and defense equipment. For anonymity reasons, there is no relation between the names of the organizations and the order of the industries. The survey consisted of a short questionnaire containing several questions about the use and sizes of designs, the used tools, and a number of configuration management functions. The questions were answered by one or two senior electronics designers from each organization. Where necessary, we held interviews with the software configuration managers of the organizations to supplement the answers from a software view.

The questions were divided in two categories: background questions and configuration management questions.

- Background questions. To gain insight in the use and size of configurable hardware, we asked direct questions on the following subjects: the applications of configurable hardware, the main device suppliers, the application of System-on-Chip designs, the number of configurable hardware developers and their background, concurrent development of configurable hardware, the estimated number of lines of HDL code, and the reuse of code for different designs.

- Configuration management questions. To understand the configuration management situation, we asked for the used tools for source version management for hardware designs and software code, for change management and product data management, the use of scripts for development and build automation, and the use of dynamic updates for configurable hardware, i.e., after production of the electronics. We also asked how system configuration baselines are established and can be retrieved.

The results of the survey are summarized in Table I. The situations at the different organizations are very similar. We discuss the results in more detail below.

**Background.** Configurable hardware is mainly used for domain-specific functionality, such as image processing. Xilinx, Altera, and Lattice are at the time of writing this article the main suppliers of configurable hardware devices. The number of developers working on configurable hardware ranges from 5 to 25, and the number of concurrent developers ranges from 1 to 6. Their background is mainly hardware, but some developers also have a software background. In particular for SoC designs, software knowledge is used, as these are developed in close cooperation with software developers. SoC designs are not common in the investigated organizations, but these are expected to be developed in the near future. Regarding the lines of HDL code, it turned out to be difficult to give an estimate on the code size; this metric is probably not as common as in software development, and the widespread use of graphical design tools will also play a role. Hardware developers are more likely to talk in terms of gates or cells of a device. Therefore, the numbers in the table on lines of code tell more about what the developers think the sizes are, rather than what the actual situation is. Reuse is done but usually in an ad hoc way, which means that code is reused but is not managed as common code and evolves independently. This can cause additional work when problems must be solved in the independently evolved common code.

**Configuration management.** Configuration management is usually about tools. Source version management for configurable hardware designs is often done with different tools: The development versions are managed with a version management tool, and the production versions are stored in the product data management system that are used for the electronic hardware. The used version management tools are some of the earlier tools, such as RCS [Tichy, 1985] and CVS [Cederqvist, 2008; Grune, 1986]. Source version management for software was done with yet another tool. For software, the mainstream version management tools are used, such as ClearCase [IBM, 2008a] and CM-Synergy [Telelogic, 2008]; the former is used in most of the organizations. The version management tools for software are mature and versatile tools. The change management tools for configurable hardware were quite different, ranging from manual MS-Excel [Microsoft, 2008] sheets to mature tools such as ClearQuest [IBM, 2008b]. Both ways seem to have served the purpose up until now. The degree of automation during configurable hardware development, such as scripting and automatic builds, is low, but the benefits of automation are recognized in most of the organizations. Several organizations are considering an automated daily build for configurable hardware. All organizations make use of dy-
Table I. Configuration Management Survey Results for Configurable Hardware in Six Organizations

<table>
<thead>
<tr>
<th>Organisation</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Background</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application of configurable hardware</td>
<td>Control Image processing</td>
<td>Various functions</td>
<td>Control Glue logic</td>
<td>Signal processing Glue logic obsolete ASICs</td>
<td>Image processing Control ASIC prototyping</td>
<td>Image processing Data routing Bus interface</td>
</tr>
<tr>
<td>Configurable hardware device supplier</td>
<td>Altera Xilinx Lattice</td>
<td>Xilinx</td>
<td>Altera Xilinx Lattice</td>
<td>Altera Xilinx</td>
<td>Xilinx</td>
<td>Xilinx</td>
</tr>
<tr>
<td># Developers(^a)</td>
<td>25</td>
<td>5</td>
<td>20</td>
<td>15</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td># Concurrent dev.(^b)</td>
<td>1-6</td>
<td>1</td>
<td>1-5</td>
<td>1-2</td>
<td>1-2</td>
<td>1-3</td>
</tr>
<tr>
<td>Developers’ background</td>
<td>hardware/electronics</td>
<td>hardware software</td>
<td>hardware some software</td>
<td>hardware, software for SoC</td>
<td>hardware</td>
<td>hardware</td>
</tr>
<tr>
<td>Lines of code HDL(^c) currently maintained</td>
<td>unknown</td>
<td>unknown</td>
<td>30KLOC for large designs</td>
<td>1MLOC (estimated)</td>
<td>100KLOC (estimated)</td>
<td>1MLOC (estimated)</td>
</tr>
<tr>
<td>Reuse of HDL code</td>
<td>yes</td>
<td>&lt;20%</td>
<td>no</td>
<td>30-50%</td>
<td>ad-hoc</td>
<td>ad-hoc</td>
</tr>
<tr>
<td>Application of SoC</td>
<td>expected soon</td>
<td>no</td>
<td>expected soon</td>
<td>yes, many</td>
<td>expected soon</td>
<td>expected soon</td>
</tr>
<tr>
<td><strong>Configuration management</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source version mgt for configurable hardware</td>
<td>RCS based</td>
<td>TeamCenter</td>
<td>RCS TeamCenter</td>
<td>TeamCenter</td>
<td>shared disk some CVS, SAP</td>
<td>CVS SAP</td>
</tr>
<tr>
<td>Source version mgt for software</td>
<td>ClearCase</td>
<td>ClearCase</td>
<td>CM-Synergy</td>
<td>PVCS</td>
<td>ClearCase</td>
<td>ClearCase</td>
</tr>
<tr>
<td>Change mgt for configurable hardware</td>
<td>manual MS-Excel</td>
<td>TeamTrack</td>
<td>TeamCenter</td>
<td>TeamCenter</td>
<td>ClearQuest</td>
<td>ClearQuest</td>
</tr>
<tr>
<td>Product data mgt</td>
<td>SAP</td>
<td>TeamCenter</td>
<td>TeamCenter</td>
<td>TeamCenter</td>
<td>SAP</td>
<td>SAP</td>
</tr>
<tr>
<td>Use of scripts</td>
<td>yes many</td>
<td>little</td>
<td>little</td>
<td>little</td>
<td>ad-hoc</td>
<td>build scripts, ad-hoc</td>
</tr>
<tr>
<td>Automated build/test environment</td>
<td>no</td>
<td>some</td>
<td>not yet</td>
<td>not structural</td>
<td>no, but desired</td>
<td>semi-automatic testing</td>
</tr>
<tr>
<td>Dynamic update of configurable hardware</td>
<td>yes, but not all</td>
<td>yes, but only for new products</td>
<td>yes, but not all</td>
<td>yes, but not all</td>
<td>yes, but not all</td>
<td>yes, but not all</td>
</tr>
<tr>
<td>Establishment of system baselines</td>
<td>Data in several archives</td>
<td>Configuration data stored in main software</td>
<td>TeamCenter and project data</td>
<td>Data in several archives</td>
<td>Data in several archives</td>
<td>Documented baselines in MS-Excel</td>
</tr>
</tbody>
</table>

\(^a\)The developers do not work full-time on configurable hardware.
\(^b\)Developers that work concurrently on the same hardware design.
\(^c\)These are very rough estimates.
namic updates for configurable hardware; hence, the device can be updated after initial production of the electronics. As we will discuss in our article, this flexibility poses additional challenges for configuration management. Finally, the establishment and retrieval of system configuration baselines is not trivial in most of the organizations. There is no automated integration between the different tools, which requires manual synchronisation. To retrieve a valid baseline, the configuration data on hardware and software configurations must be consulted in different archives.

Our experience and the results of our short questionnaire indicate that in most organizations the configuration management situation for configurable hardware is quite similar. Mainstream configuration management tools are used, but there is no automated integration between them. This is prone to errors. As such, there is room for configuration management improvement. All of the probed organizations acknowledged the need for improved configuration management for configurable hardware and system configurations, and some were already planning improvement activities in that area. But we believe that those improvement activities must be initiated together with the software world, to fully benefit from the knowledge and experience of more than 20 years of software configuration management.

In the discussions with the organizations, it turned out that the hardware world recognized that their challenges can be supported by the SCM concepts from the software world, but these were not yet commonly applied.

3. A GENERIC DEVELOPMENT CYCLE

From our interaction with the various organizations and our own experience, we derived a generic development cycle to explore configuration management on the hardware/software boundary in electronics development. We include the development, production, and service phase of a product and tie those phases to the specific configuration management aspects. The generic development cycle is based on our experience and exploration of the six organizations from Section 2.3. We provide a short introduction on how software, hardware, and system level configuration management typically is supported by SCM and PDM systems. Using a case with real-world electronics, we describe configuration management on the hardware/software boundary, including fixed and configurable hardware, software, and system configurations.

3.1. Introduction

At a typical embedded systems manufacturer, the number of software developers has outnumbered the number of hardware developers in their development department. Software and hardware development is often performed at geographically dispersed locations, and many parts are supplied by internal or external suppliers. Product variation is commonly offered through several product lines that are configured with various options.

The software in those products ranges in the millions of lines of code, and involves a mixture of the technologies and languages from the past decades. For software CM, a number of mature and versatile tools exist. To tie the tools together and tailor the process support, homegrown scripts are used for incremental building and promotion to more mature lifecycle statuses. Released software is linked with hardware through a PDM system, which contains a reference to a software baseline in the SCM system.

The hardware in a single product may consist of thousands of mechanical and electrical hardware parts. For each part, the production or acquisition process must be documented. Released hardware configurations are modified by upgrade packages, which can require software updates as well, to support the upgraded hardware in turn. Electronics and configurable hardware is being developed at the traditional hardware department. The developers design and construct the electronic printed circuit boards on which the configurable hardware devices are located. A single board may contain several devices, each ranging in complexity from about 100 lines of code to 10,000–100,000 lines of code. The configurable hardware devices typically are CPLDs and FPGAs, and VHDL and Verilog are the main languages for describing the devices. While configurable hardware is usually developed by hardware developers, the software in an electronic component (think of microcontroller code) is often the domain of software developers. The resulting electronic assembly integrates baselines from both the software and hardware disciplines.

An important classification of electronic devices is whether the software and hardware of the device can be updated dynamically or statically (by hand). A dynamic update means that the update is performed by automatic downloading of new versions from the main system software. A static update involves manually reconfiguring a device or replacing the entire assembly containing the device. Whether a device can be updated dynamically or statically depends on the design decision, not on a particular device type. Some devices simply cannot be updated dynamically, as they are not connected to
the main system software. In both cases, an initial version of the load module must be loaded for minimal testing purposes during production. After production and delivery of a product, the load modules can be upgraded in the field through a main system software update or by updating or replacing an entire electronics assembly. As we will see shortly, the distinction between dynamically and statically updating the load module into a device is very significant for configuration management, because the former is generally treated and configured as software, whereas the latter is generally treated and configured as hardware.

Configuration management is a discipline that relies heavily on tools and procedures. For software development, mature configuration management tools are required to manage the software’s inherent flexibility and complexity. Configuration management practices for configurable hardware have often evolved on an as-needed basis, and are not as mature as software CM. Early CM maturity starts with storing sources locally on developers’ disks, but this hinders tracing the source of a released load module and cooperation among developers. The situation is improved if a shared disk repository is used, where sources are stored and retrieved, and accessible by every developer. Version management is done by copying and renaming directories.

This improved situation still exists in organizations, but has severe drawbacks when several developers are working with the same files, and integrity of the repository is at stake. Variants cannot be managed properly, and collaboration and reuse of code is hindered. The use of a version management system, such as the Concurrent Versions System (CVS) [Cederqvist, 2008; Grune, 1986] or SubVersion [Pilato, 2004] can improve the situation, but the use of mature version control systems for configurable hardware sources is not widespread. In addition, the release of load modules is often very informal. The use of uncontrolled email or floppy disks is still common in some organizations. Although an informal release reduces the time to hand over and test a load module, it hinders the tracking of errors that are found during testing. Then, it is difficult to refer to the version that was released, and what its status was, which results in problems when errors must be fixed. More seriously, when the problem appears in released products, external regulations come into play. A manufacturer can be forced to hold its production or even withdraw existing products; the subsequent consequences for the business are severe, as we pointed out in the introduction of this article.

3.2. Electronics Development

Here, we describe a generic development process for a system containing electronics and software. Figure 4 gives a global overview of the development flow for software and fixed and configurable hardware development, indicated by SW, FHW, and CHW. The flow depicts initial development until installation and update at a customer, including a problem report and a subsequent system fix. We consider software development to be twofold, namely, software in the fixed hardware assembly and main system software that performs updates of the software and hardware in the fixed hardware assembly. In the figure, the former is indicated by the software development line, whereas the latter is indicated by the system development line.

The goal of the figure is to illustrate the different ways of releasing during the product lifecycle, which clearly depends on the particular lifecycle phase. In fact, the initial release for production involves both hardware and software production, whereas the system fix involves only a release to software production. The flexibility of configurable hardware and software allows for a shorter time-to-market of the fix. The figure illustrates the complexity of the configuration relations, as configurable hardware and software are released within and across the disciplines.

Development starts at the system level, from where the design is partitioned into subsystems and into hardware and software. This is indicated on the left-hand side of Figure 4. For an initial product version, software and fixed and configurable hardware is baselined and released towards hardware production for production tests. This is indicated in the figure by the solid line from System development towards Hardware production. Software and configurable hardware is also baselined and released towards Software production to allow installed systems to be updated with new versions. This is indicated by the solid line from System development towards Software production. Then, the product is produced and installed at the customer; see “Install at customer” near the middle of the figure. After installation at a customer, a problem is reported. The problem is solved in configurable hardware and software; hence, no release towards hardware production is required for this type of update. The updated software and configurable hardware is released, and the previous versions are updated at the customer through the main system software. A prerequisite is that the configurable hardware can be updated dynamically. Statically updateable hardware follows the path through hardware production, and a new version of the fixed hardware assembly containing the updated configurable hardware device is installed at the customer. The flexibility of software and
configurable hardware reduced the time to solve the problem. Likewise, it also allows for a shorter time-to-market for new features.

To illustrate the development process, we describe a case with electronics through its different life-cycle phases.

**Case: MoveAndMonitor (MAM) electronics.** To illustrate our generic development cycle, we use a case with real-world electronics from one of the investigated organizations. This case is considered to be generic; i.e., we believe it is exemplary for many electronics development situations. For anonymity reasons, we cannot explain the function of the electronics in detail.

The electronics is a hardware electronics that is contained in a larger mechatronic subsystem, and its main function is to perform the movement and monitoring of the mechatronic subsystem. We will refer to the electronics as the MoveAndMonitor (MAM) electronics. The larger mechatronic subsystem itself is also part of a larger system, containing additional hardware and software. We refer to the mechatronic subsystem as the “subsystem” and the larger system as the “main system.” The MAM electronics interfaces with the main system, which sends movement instructions for the subsystem to the MAM electronics. During movement operations, the MAM electronics uses several sensors to monitor the position, angle, and speed of the subsystem.

**Specification.** Any development effort should start with the specification of requirements. For the development of electronics, which is usually part of a larger system, its requirements and interface with the rest of the system must be specified. Hence, for the MAM electronics, the starting point was a requirements/interface specification, describing its overall functionality and its interfaces. The requirements were derived from the specifications for the main system and subsystem.

**Design and implementation.** Using the specifications for an electronic assembly, the design and implementation can commence. This results in a functional design schematics. The next step is to actually implement the functionality: for fixed hardware a physical
assembly implementation, for configurable hardware a design for the configurable hardware devices, and for software a software program implementation.

For the MAM electronics, the development started with a functional design of the electronics schematic, specifying its functions. The schematic is a multi-level block diagram that gives a global overview of the board in terms of functions and interfaces. Figure 5 shows the top-level schematic for the MAM electronics. It consists of seven functional blocks; we briefly describe them starting at the top-left and ending at the bottom of the figure:

- MEMORY is for volatile and non-volatile storage of data.
- PROCESSOR is for processing movement instructions.
- FTCBUS is an optional interface for additional sensor functionality.
- COMMUNICATION is for communicating with the main system.
- TWINDRIVE contains logic for controlling the movement in two directions.
- POWER is for power management and conversion.
- CONNECTOR is for physically connecting the electronics to the mechanical subsystem for power, communication, and signal filtering.

The functional blocks are interconnected by signals. Each function is detailed in separate schematics, which can be found on the pages that are indicated for each block. The number of pages for each component can be seen as a complexity indicator. For instance, the TWINDRIVE function consists of 26 pages, while the MEMORY function consists of only 2 pages.

Using the specifications and the design schematic, the detailed physical outline of the MAM electronics was developed. The functional blocks were mapped onto a printed circuit board assembly with actual hardware parts with fixed and configurable hardware devices. It was decided which functionality had to be implemented in fixed or configurable hardware, or in software. The hardware/software interface defined the physical connections in terms of physical pin mappings; the hardware/software interface defined the interface between a board and software, in terms of memory mappings. The real-time processing of the sensor inputs during movement was implemented in an FPGA for performance reasons. Tasks that did not require high performance, such as the communication with the main system, were implemented in software running on an off-the-shelf CPU. A real-time operating system—VxWorks [WindRiver, 2008] in this case—supported the overall infrastructure. The use of software and a CPU shortened the development time and also reduced costs, as the CPU was cheaper than an FPGA that would be required for implementing the software functionality. For booting the CPU and instantiating and updating the FPGA, a CPLD provided the necessary functionality. The design of the actual assembly layout was done in parallel with the development of the software and the configurable hardware device. The software was programmed in the software languages C and C++, whereas configurable hardware devices were designed in the hardware description language VHDL. The hardware and software were compiled into load modules, which were subsequently stored in the non-volatile memory of the MAM electronics.

To illustrate the actual implementation, we discuss a VHDL code excerpt for the FPGA. The code excerpt displays the parameters for the function that implements the retrieval of board and load module configuration data from the CPLD and storing it in the FPGA; see Figure 6. This data is transported serially, to limit the number of connections between CPLD and FPGA. The code specifies the mapping between signals, input/output pins, and registers that are necessary for retrieving the board identification and version of the CPLD load module. The function for retrieving the data is the entity reg_update (line 1); an entity is comparable to a function or method in other languages. The entity is configured to support one transmit register and 6 receive registers, all 8 bits wide (lines 2–3). The reset and clock signals are required for resetting the entity and for the timing of the (synchronous) implementation (lines 5–6). The sout signal is bound to the input pin of the CPLD, whereas sin is bound to its output pin (lines 7–8). Note that the code specifies the destination of the FPGA signals towards the pins in the CPLD. Lines 9 and 10 specify the details of the transmit register; the lower bit is used for a standby signal that can be send to the CPLD and the rest unused and filled by zeroes. Lines 11–13 specify the receiving registers for the board identification and the major and minor version of the CPLD load module; as stated, there are 6 registers each 8 bits wide, which accumulates to 48 bits in total. The actual configuration data is used during start-up of the system to identify the electronics, as the board identification is provided by the CPLD. This is further discussed below.

Figure 7 shows a drawing of the MAM printed circuit board assembly, with its actual layout. The layout consists of several layers of wires; the figure shows the top layer only, with the main hardware/software parts identified: CPU, FPGA, CPLD, volatile and non-volatile memory and communication with the main system software. A large amount of space is consumed
Figure 5. Design schematic for the MAM electronics (for details magnify the electronic version of this article with your PDF reader).
by the power conversion—the two large boxes in the center—and by the physical connectors at the bottom of the assembly. The drawing has been added to show the results of such a design endeavor. Our scope is to discuss the configuration management aspects of how to get there.

System start-up. At system start-up, the electronics assembly is identified and initialized by the main system. This is usually done through instructions in the software load module, which are loaded into volatile memory, and from there the electronic assembly starts its operation. Those instructions include some to load the hardware load modules from the memory into the logic circuit of the configurable hardware device, thereby instantiating the hardware device with its behavior. After the instantiation, the main system software verifies that the device has the latest version. If not, the version in the nonvolatile memory is updated and the new version of the load module is loaded into the device.

Release for system test. Before a load module can be integrated formally, it must be released. The phase of the development cycle determines the degree of formality that must be applied for a release. In a prototyping phase, when the change rate is high, the degree of formality is low. The main focus is on proper identification and status of released load modules; i.e., make sure that its internal version identification is consistent with its external marking. When the prototype matures, more formal change control is applied. The release of a load module depends on how the load module is loaded into the device. The load process is either dynamic, i.e., through a main system software update, or static, i.e., loaded by hand. The release process of the load modules depends on whether the device is loaded dynamically or statically. In the prototype phase, when the electronics assembly is not yet integrated with the main system, both types of devices are updated by hand.

As there exist hardware and software load modules, a possible scenario is that both hardware and software developers own a physical prototype to perform their tests. Every now and then, the hardware developers release a hardware load module towards the software developers, and software developers release software load module towards the hardware developers. Each developer then loads the latest version into their prototype to perform tests. When the prototype matures and is released itself, dynamically loadable modules must be updated through main system software updates.

Dynamically loadable devices. The release and integration for dynamically loadable devices involves the release of a load module for integration with the main system software. The load module is incorporated into the current main system software build. From there, the load module is loaded into the memory device during start-up of the main system. New versions of load modules are released through main system software releases or patches. As there exist different versions of the hardware during development and in the field, the main system software needs to know which versions of load modules and hardware are compatible to perform updates on released products. As we will show next, the version relation must be defined.

To explain the configuration relations, we show the configuration data of the MAM electronics in Figure 8, which is stored in a plain text file and readable by the main system software. Most devices of the MAM electronics are dynamically loadable. In the figure, two versions of the MAM printed circuit board are shown: MAMBoardV4 and MAMBoardV5. Boards are identified with a major and minor identification (lines 3–4 and 27–28). For each board version, the compatible versions of the hardware and software load modules are shown. Each version of the MAM board has three different load modules: the APPL module that contains the operating system and some application code, and the FPGA that contains the hardware design for the FPGA. Load modules are also identified with a major and minor version number (lines 10–11, 19–20, 34–35, and 43–44). The load module itself is identified by the File field, which is the actual file on the main system’s file system. The load modules for both board versions are similar except for the load modules for the FPGA. Load modules are also identified with a major and minor version number (lines 10–11, 19–20, 34–35, and 43–44). The load module itself is identified by the File field, which is the actual file on the main system’s file system. The load modules for both board versions are similar except for the load modules for the FPGA. Board version 4 requires FPGA load module version 3.2 (lines 19–20), whereas board version 5 requires FPGA load module version 3.9 (lines 43–44).

Upon the release of a new version of a MAM board or a load module, the configuration data file must be updated to reflect the supported configurations.

Statically loadable devices. The release and integration for a statically loadable device involves manual loading the load module into the memory of a board.
Figure 7. A drawing of the MAM printed circuit board assembly with the main hardware/software parts identified (for details magnify the electronic version of this article with your pdf reader).
Appropriate identification must be applied, i.e., the identification of the new load module version is marked on the board, or this relation is stored in a proper place. Some of the functionality of the MAM electronics was implemented using a CPLD, which was statically loadable. The load module for that device must always be updated by hand.

**Release for production and service.** For production, the load modules must be linked to an assembly. Using the board data, a board supplier produces a board and loads an initial version of a load module onto the board. After the supplier’s production tests, the boards are shipped to the systems manufacturer. From there on, the production process differs for the two ways of loading a configurable hardware device.

For statically loadable devices, boards are further tested at the systems manufacturer and then delivered to a customer. For installed products, a service engineer inserts an updated board into the installed product.

For dynamically loadable devices, load modules are stored in the main system software and released with a main system software release. For newly ordered products, load modules are distributed with a main system software release. If the initial version that was loaded at the board supplier must be superseded, it is dynamically updated during system start-up. Subsequent changes to installed products are performed by main system software updates containing new versions of load modules. Note that a replaced board, for example, due to a defect, may also require an update of a load module, as the board can be replaced by a newer version that requires a new version of a load module.

Dynamic updates can cause various problems. In one of the organizations, we encountered an example where a dynamically updateable printed circuit board was considered broken at a customer installation. When the board was returned to the supplier, the supplier simply restored the original load module that was supplied for production, successfully performed the board’s tests, billed the systems manufacturer for the repair, and re-sent the board. When the board was reinstalled at the customer, the load module was again dynamically updated by the main system software, and the original error reoccurred. When the supplier was confronted with this issue, the response was: “The product was modified after we delivered it, and we were helpful in bringing it back to its former condition.” In the end, it turned out that the error was caused by the updated load module. The example illustrates that the responsibilities for different types of errors must be clearly defined.

Figure 8 shows part of the bill-of-materials of the MAM electronics, which is used for production purposes. The list is extracted from a PDM system and shows a number of fixed and configurable hardware components with their assembly level, identification number, description, and quantity. The top level fixed hardware assembly is the MAM electronics, which consists of a number of fixed hardware components and two configurable hardware devices, a CPLD and an FPGA. Two components, named “PRIC,” are assemblies themselves. PRIC stands for PRogrammable Integrated Circuit, and these assemblies are in fact placeholders for the actual load modules, source files, and a configurable hardware device or a memory chip that stores the load modules. The upper PRIC contains the load module and sources for the CPLD, and the CPLD itself; the load module is stored in the nonvolatile memory of the CPLD. The lower PRIC contains load modules and sources for the CPU and FPGA, and a
Figure 9. Part of the bill-of-materials for the MAM electronics, extracted from a PDM system.
flash memory; the load modules are both stored in the same flash memory.

Note that the sources for the load modules were stored in the bill-of-materials as well, but these can soon become outdated when the load module is updated after production of the board.

Summary. We presented a generic development cycle for electronics and explained the configuration management aspects that come with the hardware/software boundary. In particular, there are intertwined configuration relations that cause significant complexity. Next we will describe these relations in more detail.

3.3. Configuration Management

As we have seen in the previously described development cycle, there exist intertwined configuration relations for hardware and software. Those relations must be properly established and managed using SCM and PDM systems.

Product structure. Figure 10 gives a simplified view on a typical product structure for an electronics product. The figure shows a fixed hardware assembly containing configurable hardware and software, in combination with a single main system software release. Each entity is stored in an SCM system, a PDM system, or in both systems. The valid hardware and software configurations are stored in the configuration data entity; we showed an example of such data in Figure 8. Note that earlier versions of the software and configurable hardware load modules, indicated by dashed boxes, must be present in the current main system software baseline, to support previous hardware versions; we discuss this below. A solution to traceability between sources and load modules is to store the sources in the PDM system as well. We found this in some of the investigated organizations. However, only the sources of a version that was released at the hardware release for production is archived; hence, these sources may turn out to be misleading when a field problem must be solved but the load modules have been updated dynamically after production.

Baselines and configuration models. Software and configurable hardware are stored and baselined in SCM systems. Fixed hardware assemblies are baselined in PDM systems. Hardware and main system software are also baselined at the system level in PDM systems to define a system-level configuration. A reference from

Figure 10. A simplified view on the product structure of a product containing software and electronics, and the relation with SCM and PDM systems.
the PDM systems identifies the current main system software release. Several versions of the same load module are included in the system level baseline; the hardware assembly has an initial version of the load module for the production test, and the main system software contains a newer version of a load module that is loaded during start-up. In addition, the main system software must contain previous versions of load modules to remain backwards compatible with previous fixed hardware versions that have been installed in the field. If these installed systems are upgraded with a new main system software release, it should be backwards compatible with the fixed hardware, otherwise the fixed hardware must be upgraded as well. The configuration data, which is stored and baselined in the main system software, provides the essential information for updates. Hence, the valid system configuration for hardware and software is actually contained in the main system software baseline. We illustrate this with an example scenario in Figure 11, where a number of system releases are shown in combination with the component versions. The valid system configurations are as follows:

- System release R1: (FHW v1, CHW v1, SW v1)
- System release R2: (FHW v1, CHW v2, SW v2)
- System release R3: (FHW v1, CHW v2, SW v2) (FHW v2, CHW v3, SW v3)

The first release R1 contained from each component the first version. In the subsequent release R2, both software and configurable hardware released a new version; these were backwards compatible with the previous fixed hardware version. In release R3, a new fixed hardware version component was released, which required new software and configurable hardware versions. At the same time, it was decided that release R3 should be backwards compatible with the previous hardware; hence, the previous configurable hardware and software were incorporated in R3. This allows an installed system to be upgraded to release R3, without having to change the fixed hardware. Note that the number of valid system configurations will continue to grow until certain hardware versions are declared obsolete. As the lifecycle of products containing electronics can span up to over 10 years, obsolescence may require installed systems to be upgraded. Therefore, it is essential to have a proper configuration model in order to make cost-effective decisions on configuration changes.

**Statuses.** Another aspect is the promotion statuses for hardware and software, which can differ as well; Table II gives an example of such statuses. Software has generally more statuses than hardware, which is probably due to the iteration frequency. It starts at status built and ends with status end-of-service. The first formal status of fixed hardware is prototype; its end status is obsolete. Configurable hardware is somewhere in the middle; it has statuses from both software and from hardware. The applied status can also depend on whether a load module is statically or dynamically loadable. It may take a while before dynamically loadable load modules enter the status obsolete, as it must support previous fixed hardware. As long as old versions of fixed hardware can be encountered in installed systems, the accompanying load modules must be present in the main system software. In some of the studied organizations, it was not always clear which previous fixed hardware versions were still installed at customers. Therefore, many versions of load modules had to be supported in order to remain backwards compatible.

**Summary.** The investigation of the product structures, baselines, and statuses on the hardware/software boundary provides more insight in the complexity of

<table>
<thead>
<tr>
<th>Table II. An Example of Promotion Statuses for Software and Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
</tr>
<tr>
<td>built</td>
</tr>
<tr>
<td>tested</td>
</tr>
<tr>
<td>integrated</td>
</tr>
<tr>
<td>alpha</td>
</tr>
<tr>
<td>beta</td>
</tr>
<tr>
<td>release</td>
</tr>
<tr>
<td>service</td>
</tr>
<tr>
<td>pack</td>
</tr>
<tr>
<td>end-of-service</td>
</tr>
</tbody>
</table>

![Figure 11. An example scenario for valid system configurations.](image)
combined hardware and software CM. We showed that development is performed across different organizational units, different processes are used, and different tools are employed. This requires integration or proper alignment of the organization, processes, and technology. In the next section, we discuss this in more detail.

4. DISCUSSION

The previous sections showed that the hardware/software boundary contains significant complexity. To enable proper evolution, the hardware and software are brought together by a better alignment in several ways. In this section, we discuss the challenges that appear when one attempts to align those worlds; we use our experiences from the six investigated organizations for illustration purposes. This section does not provide off-the-shelf solutions, but is rather intended to raise the awareness on the issues that are encountered when hardware and software worlds must work together.

Our discussion is structured into three areas: organizational, process, and technical alignment. Although these areas are interconnected and should not be approached in isolation, we believe that the focus must be on processes and technology alignment, as this will provide most benefit to the development process on the hardware/software boundary.

Organizational alignment. From an organizational perspective, software, and hardware engineering does not need to be managed very differently. Although hardware engineering involves logistical aspects that influence the way of working more than in software engineering, the same important organizational issues are present both within and across the disciplines: people need to communicate and work together. The organizational issues that arise when hardware and software developers must work together originate mainly from the different terminology in both worlds. We have encountered several situations where cooperation between hardware and software people was severely hindered by misunderstandings due to different terminology. Those misunderstandings also support the perception that CM in one discipline is more mature than in the other. The creation of a single language for both worlds in an organization is an important alignment step towards increased productivity.

Another alignment step is the deployment of cross-functional organization units, consisting of software and electronics developers. Cross-functional teams enable and improve communication on the hardware/software boundary. This allows the developers to better understand each other and work together towards a common result. Cross-functional project teams for software and electronics were found in several of the investigated organizations, but the traditional hardware/software departments usually define the organizational boundaries, which results in different processes, tools, terminology, and ways of communication. These differences hinder the productivity and product evolution on the hardware/software boundary. The acknowledgement and awareness of those differences is the first step towards improved alignment.

Process alignment. Another area of alignment is the process. The software development process is focused on quick iterations with frequent baselines and releases. Hardware development is more dominated by lead times of production, installation, and configuration of physical objects. Software that is on the hardware/software boundary needs to take the hardware lead times into account. Formulating a common development process overview with common integration baselines, such as we showed in Section 3, helps the two worlds to connect and understand each other. It also helps to illustrate the distinct aspects of each discipline with regard to the release moments and production phases. Also, a common process promotion model aligns hardware and software statuses where necessary. We showed an example of promotion statuses in Section 3. Although there are different statuses, such as built for a successful built software component and prototype for prototype hardware, common statuses already exist, such as released for integration and production. By aligning statuses, developers from both disciplines have a better knowledge of the status of each others' work, and use this to improve planning their own work.

Another step to align hardware and software is to standardize the change management process. We have observed situations where a problem report was sent back and forth between hardware and software developers, because both of them were confident that the problem was caused by the other discipline. A unified change management process, where proposed changes on the hardware/software boundary are evaluated by a control board with sufficient expertise of both disciplines, aids to avoid such situations.

An additional important role of processes is to fill the integration gaps between different tools. We discuss this issue in the technology alignment below.

Technical alignment. In the previous section, we have seen the intertwined relationships on the hardware/software boundary, and the usage of SCM and PDM tools to support the distinct development processes. The alignment of technology is achieved by aligning SCM and PDM tools. These tools ultimately form the connection between the hardware and software world from an information management point of view. The integration of SCM tools for software and PDM
tools for hardware is a research subject in itself. We have also seen that for configurable hardware mature SCM tools are needed as well, to support the growing complexity and concurrent development of hardware designs. Hence, there are several tools that have dependencies with each other, and integration of those tools minimizes information management mistakes.

In the investigated organizations, besides the integration of the development environments and the configuration management tools, we found no automatic integration between the SCM and PDM tools. To retrieve accurate configuration data, information from several information systems had to be consulted and combined. This is possibly caused by lack of integration for the specific tools. However, for several combinations of SCM and PDM tools, vendors do provide off-the-shelf integration solutions. Hence, these solutions either do not have all the required functionality, or the industry is slow in adopting them. In the latter case, it is only a matter of time before several of the challenges have been solved. In the former case, it could be a judicious choice not to integrate the tools, in order to prevent the tools from intertwining and growing into each other. The tools will become difficult to maintain, while they were never intended to be integrated when they were initially designed. It was also observed by other researchers that the integration is complex, as it requires considerable knowledge of several technologies; hence, many end users are not capable of performing the integration alone [Crnkovic, Asklund, and Dahlqvist, 2003: 148]. Therefore, it is not attractive to build homegrown integration tools, as changes to the involved technologies cause a maintenance burden on the integration tools. A loosely coupled integration that is supported by a proper process can be sufficient to support the development process. Nevertheless, SCM and PDM clearly have intertwined dependencies with each other, in particular, on the hardware/software boundary. Although steps are taken towards standardized tool interfaces and integration (e.g., through standardized XML interfaces), these are not yet widely adopted by the industry. Until then, it is important to acknowledge the existing dependencies between SCM and PDM, and to decide to what extent automated tool support is beneficially.

5. CONCLUSIONS

More flexibility in hardware configurations and more overlap between hardware and software components require changes in the way system development organizes configuration management. In this article, we analyzed several situations from the embedded industry to raise awareness and make a step forward in this challenge. We discussed configuration management on the hardware/software boundary. We proposed a generic development cycle that we generalized from studying six large organizations. Based on the generic cycle, we modeled the development flow for the involved disciplines and discussed the configuration management situation for software and fixed and configurable hardware. We argued that alignment of organization, processes, and technology must be tailored to the circumstances. Our observations on those alignments provide input to the industry and other researchers. Although we investigated just six organizations, we believe the situations are typical; hence, both the industry and academic world benefit from our results.

We discussed the fact that the software world established a mature set of software configuration management practices; the hardware world, with its advances in configuration flexibility, benefits from those practices. On the other hand, the hardware world has established mature configuration management processes a long time ago. Although it is not the focus of this article, the software world may also benefit from the hardware practices. For instance, the concept of interchangeability is firmly established in hardware to manage configurations and to increase flexibility; it will also provide benefit to component-based software development.

Indeed, the two worlds are quite disconnected due to various differences, such as different design flows and available tools but also the different cultures and histories. Both worlds have evolved in isolation, and when

<table>
<thead>
<tr>
<th>Table III. Acronyms Used in This Article</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASIC</strong></td>
</tr>
<tr>
<td><strong>CAD</strong></td>
</tr>
<tr>
<td><strong>CHW</strong></td>
</tr>
<tr>
<td><strong>CM</strong></td>
</tr>
<tr>
<td><strong>CPLD</strong></td>
</tr>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td><strong>ECU</strong></td>
</tr>
<tr>
<td><strong>(E)PROM</strong></td>
</tr>
<tr>
<td><strong>FHW</strong></td>
</tr>
<tr>
<td><strong>FPGA</strong></td>
</tr>
<tr>
<td><strong>HCM</strong></td>
</tr>
<tr>
<td><strong>HDL</strong></td>
</tr>
<tr>
<td><strong>HW</strong></td>
</tr>
<tr>
<td><strong>HWiL</strong></td>
</tr>
<tr>
<td><strong>MCU</strong></td>
</tr>
<tr>
<td><strong>PRIC</strong></td>
</tr>
<tr>
<td><strong>PDM</strong></td>
</tr>
<tr>
<td><strong>SCM</strong></td>
</tr>
<tr>
<td><strong>SOC</strong></td>
</tr>
<tr>
<td><strong>SW</strong></td>
</tr>
<tr>
<td><strong>VHDL</strong></td>
</tr>
</tbody>
</table>
we consider the past, we expect it to stay that way for some time to come. Although there are no silver bullets here, to enable efficient system evolution, organizations must move interdisciplinary configuration management to a more mature level. This starts at the organizational level and is supported by aligned processes and tools. The hardware and software tool providers must move further into each others’ worlds by supporting the distinct aspects of the development processes and provide clear and smooth integrations. Both worlds must cooperate to enable and support rapid and reliable changes to interdisciplinary development. Hardware and software process adjustment and the accompanying development and configuration management tools with standardized interfaces are some of the most important prerequisites to enable efficient system evolution.

ACKNOWLEDGMENTS

This research was sponsored by Philips Healthcare, The Netherlands, as part of the CoCoCo project: Concepts to Control Configurations. We thank all the involved companies for their stimulating discussions and kind cooperation. We also thank all anonymous reviewers for their valuable comments and suggestions for improvements.

REFERENCES

W. Babich, Software configuration management: Coordination for team productivity, Addison-Wesley, Reading, MA, 1996.
S.A. Dart, Parallels in computer-aided design framework and software development environment efforts, Proc Third
IFIP WG10.2/WG10.5 Workshop, in cooperation with GI/ITG FG 3.5.6/5.2 on Electron Des Automat Frameworks, North-Holland, Amsterdam, 1992, pp. 175–189.


K. Howell, Reprogrammable logic drives automotive vision systems design, FPGA Struct ASIC J (February 2007).


M. Mason, For automotive electronics, FPGAs are in the driver’s seat, Electron Des (June 2006).

C. Maxfield, FPGAs gear up for new automotive applications, EETimesAsia (October 2007).


C.G. Newborn, Analyzing the Army’s configuration management system applicability to a commercial cataloguing system, Master’s thesis, Naval Postgraduate School, Monterey, CA, 1997.


Rene Krikhaar is associate professor at the VU University Amsterdam, The Netherlands. His research topic concerns Collaborative Software Production. Software intensive systems are more and more developed by different organizations at multiple development sites. The proper interaction between architecture, lifecycle processes, and configuration management can resolve a large portion of the problems industry is currently facing. Rene has over 20 years experience in the industry in software engineering, architecture and configuration management. In his industrial career he joined several companies in the field of telecommunication, IC design tools, healthcare in roles as configuration manager, software engineer, architect, and industrial researcher. Currently, he is also leading a consultancy company, ICT NoviQ, which provides industry working solutions and improvements in Configuration Management, Architecture, Requirements Engineering, and Development Processes.

Wim Mosterman started his professional career at Philips Medical Systems, as a hardware engineer in the Computed Tomography Development Department in 1981. He participated in the hardware development of the Tomoscan series of Philips CT scanners, most notably the development of printed circuit boards with embedded processors. In 1988 he was appointed as a Software Engineer in the Magnetic Resonance development department, also at Philips Medical Systems. Here he worked many years in the software development of multiple generations of the Gyroscan series of Philips MR scanners, his work varying from test software, platform libraries, and GUI to RTOS on intelligent peripherals developed by the MR Hardware department. In 2005 he started as a hardware designer in this same MR Hardware Department of what is now called Philips Healthcare. Next to the design of content for FPGAs (including SoC solutions), he is also responsible for the integration of Configuration Management tools in the FPGA development cycle.

Niels Veerman is an industrial researcher at the VU University Amsterdam and a process improvement consultant at embedded systems manufacturers, including Philips Healthcare. His primary interests are product design and development, and software and systems engineering, evolution, and reengineering. Niels has worked on various software and systems engineering projects, including legacy system migrations for the financial services industry and system configuration management for the embedded systems industry. He received an MSc and a PhD degree in software engineering from the VU University in Amsterdam.

Chris Verhoef is a full professor at the VU University Amsterdam. He is Vice Chair conferences of the IEEE Technical Council on Software Engineering. He served in Steering Committee, General Chair and Program Chair positions for many important juried software-related research conferences. Chris acted as an industrial consultant in several IT-intensive areas, notably hardware manufacturers, telecommunications companies, financial enterprises, government, and large service providers.