Component-oriented Version Management for Hardware Software Co-design

Tien N. Nguyen and Zhao Zhang

Electrical and Computer Engineering Department
Iowa State University
{tien,zzhang}@iastate.edu

Abstract

Nowadays, the development of modern computing devices involves a substantial and growing part of software development. A great challenge for engineers is to manage the evolution of a system with several components in the face of mounting complexity due to concurrent *hardware* and *software* development. This paper presents our preliminary research results on a novel component-oriented version management mechanism that is capable of versioning the underlying logical contents of components in system design models and associated software artifacts in a cohesive manner. We have applied this approach to create a prototype of a versioning system for hardware software co-design.

1 Introduction

Embedded computing systems have been playing vital roles in the information infrastructure of our society. They promise a great potential for many critical applications, such as on-site processing of sensor data, bio-security, and non-destructive fault detecting. A challenge during the design, development, and maintenance of an embedded system, often called *a hardware software co-design process*, is to maintain the productivity in the face of mounting complexity due to concurrent hardware and software development. Engineers need tools to manage a wide variety of artifacts that are constantly evolving.

However, the management task is still unsatisfactory. One of important issues is the poor interoperability between specialized version control tools for different types of artifacts, thus, making it difficult to maintain the logical connections among artifacts. For example, in the current practice of a hardware software co-design process, requirement and analysis specifications are often produced using Office suites such as Microsoft Word or Excel. When a new version of a specification is needed, entire new document file is created. For high-level architectural system design, they use specialized environments such as IBM Rational for UML diagrams [7] or SCE [1] for SpecC development methodology [2]. Version control capabilities of these environments vary a lot. IBM Rational uses ClearCase [4] as its main configuration management system (CM). However, ClearCase cannot support an arbitrary type of design diagram. SCE focuses on system modeling and provides no versioning support.

Low-level hardware designs are commonly managed by integrated development environments such as the Xilinx XPS (Xilinx Platform Studio) [8]. Software programming and hardware coding, plus integrating and debugging functionality are also provided. However, versioning support for hardware designs in those environments is limited to the use of *version tags* or *model names* for hardware devices. The structures of devices in those designs are not versioned at all. Versioning support for software components is provided via external text

file-based source code versioning tools such as CVS [5]. Similar to other file-based CM tools, they have no knowledge about the underlying semantics of software components. At the board design level, layout designs are maintained within CAD routing tools with little version management support.

While efforts to integrate version control tools for hardware and software components have had limited success, the main reason for the poor interoperability is the lack of a unified CM model that is able to support both hardware and software design/implementation. Limitations of existing versioning models include their inadequacy in representing semantics of design models and inability to manage versions of both hardware designs and software components in a tightly connected manner.

2 Our Approach

To address this problem in hardware software co-design, we suggest a unified, *component-oriented* approach to configuration management, in which all hardware design objects, source code, and documentation are considered to be *components*. All components (including hardware and software) and logical relations among them are put under version control. Components are directly accessible from the repository. Consistency of versions is maintained among components, rather than among files. The versioning system is capable of capturing the logical structures of components in designs and their hardware/ software artifacts.

Our goal is to provide versioning supports for any software and hardware design components produced during a co-design and maintenance process. Within our CM framework, we need to have a representation for components that can capture the semantics of components. The success of the object-oriented technology allows us to model any type of components. With an object-oriented programming language (e.g. Java), we can easily capture the semantics and logical structure of any software or hardware design components. It is apparent that a mechanism is required to provide version control for any type of object defined as a class.

That mechanism must satisfy the following requirements. Firstly, it must be able to provide version control for any *structured* components and the relationships among them since components can be composite. Secondly, the mechanism must have an explicit storage representation for changes between different component versions. Each version should not be stored as a whole entity as in existing versioning tools for hardware designs [3]. Thirdly, the differences between versions must be directly accessible in the repository, rather than to be computed using complex differencing algorithms. Fourthly, the mechanism must support *document-centric* artifacts such as programs and documentation. The states of a component including structure, content, and properties at different versions must be easily retrieved. Finally, logical connections among artifacts must be maintained.

We have built such mechanism that met those requirements. That novel component-oriented CM mechanism was built upon Molhado versioned data model and its associated CM repository [6]. The CM mechanism is *not* hard-coded to support a fixed set of component types. A new type of component (hardware design component or software component) can be defined as a Java class. Our CM library functions are responsible for providing versioning supports for any type of component. The hierarchical structures among hardware design components are often more complex than tree-based compositional structures of program entities. For example, a hardware device consists of constituent components that might be connected to each other in a certain structure according to the chosen design. To support structured and complex components of an embedded system, a *directed graph-based* version control scheme was developed. Graph-based structure is used to represent the internal structure of a component. That type of component is called *composite component*. Without internal structure, a component is called *atomic*. Our graph-based versioning algorithm also supports document-centric software artifacts such as source code and documentation.

If a third-party co-design environment is built with its own component representation, a bridge is required to act as a converter between the native representation model of that envi-

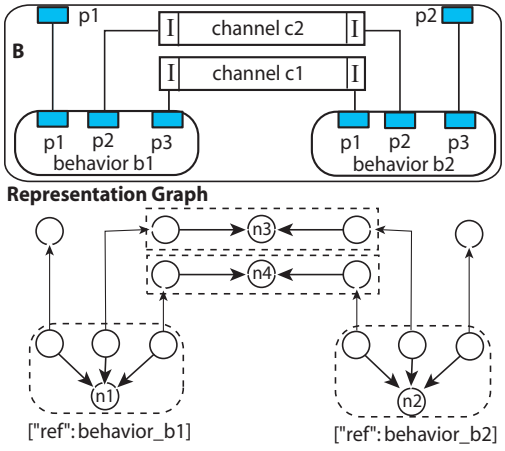

Figure 1: A SpecC "Behavior" Component

ronment and our representation model. The only requirement on the bridge is that it needs to call our CM library functions to update our repository whenever users check in changes to components. If a new tool/environment for a new type of component is built using our graph-based representation, a bridge is not needed.

3 Tool Development

Based on that infrastructure/mechanism, we have built, *EmVC*, a prototype of a CM system for hardware software co-design process for embedded systems with SpecC development methodology [2] and FPGA technology. The distinguished feature of that CM system is the cohesive versioning management among all artifacts of a hardware software co-design process including requirements, design documents, specifications, hardware designs, and associated software components. Changes are integrally captured and related to each other in a cohesive manner. This section illustrates how EmVC's components are modeled using our version management mechanism.

According to SpecC, the functionality of an embedded system is captured as a hierarchical network of *behaviors* interconnected by hierarchical *channels*. Specifically, a system can be described in terms of *behavior*, *port*, *channel*, and *interface*. A *behavior* defines functionality of a hardware or software component. It can be connected to other behaviors or channels through its *ports*. A behavior can be composite if it contains child behaviors. The functionality of a behavior is specified by a piece of SpecC code. A behavior is modeled as a *composite* component. Its connection to another is represented by a directed edge in the directed graph representing the connection structure among them. Properties of a behavior are versioned as well. An atomic component type is created to model a port, which belongs to one behavior.

A *channel* defines how the communication is performed. It corresponds to a set of SpecC variables and methods. It can be hierarchical and sub-channels are used to specify lower level communication. Similarly to behaviors, a channel is defined by a component type. Compositional structure of a channel is also represented as a tree or a graph in a composite component. An *interface* represents a flexible link between behaviors and channels. It consists of declarations of communication methods which are defined in a channel. An interface is modeled by an atomic component. Figure 1 shows an example of a behavior B consisting of two sub-behaviors b_1 and b_2 which communicate via channels c_1 and c_2. Each behavior has three own ports, and each channel has two own interfaces. B is represented as a composite component whose structure is modeled as a directed graph. The "ref" attribute associated with each node contains a reference to the corresponding component. E.g., for $n1$, the "ref" attribute refers to the component "behavior_b1". A predefined component, "product", is provided to represent entire system.

Programs are modeled as ASTs, which are represented in our tree data structure. Similarly, documentation can be considered as tree-structured documents as in XML or HTML. XML files are represented in a similar manner. Internal structure of an XML component is represented via an XML document tree.

Components in design models are logically related to each other and to software components at the implementation level. They also have interdependencies and connections with documentation and specifications. To manage logical relations, we use Molhado's hypermedia infrastructure in which a link is represented as a *first-class* entity [6]. Those hyperlinks facilitate the process of browsing, visualizing, and

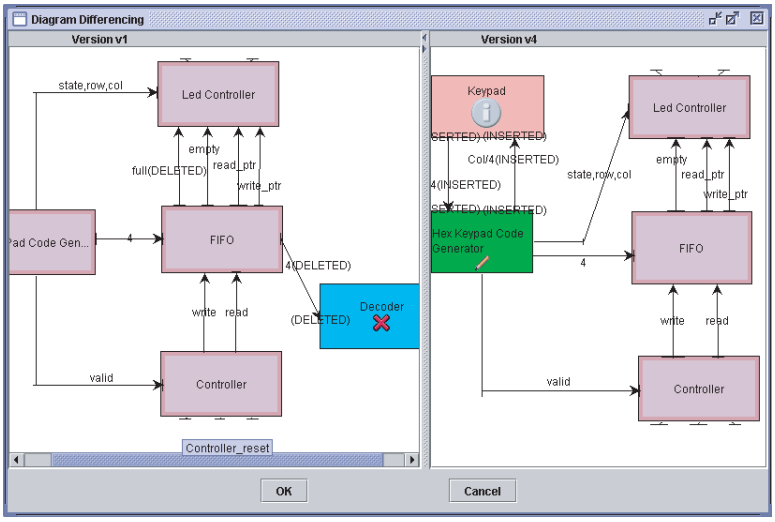

Figure 2: Differences between Versions of a System Design Diagram

analyzing of relationships among components.

Figure 2 shows changes between $v1$ and $v4$ of a diagram for a FIFO Keypad Scanner. Icons attached to graphical objects signify the nature of the changes (deletion, insertion, modification, relocation). For example, between $v1$ and $v4$, behavior "Decoder" was deleted ("X" icon), "Keypad" was inserted ("I" icon), etc.

4 Conclusions

This paper addresses the lack of CM systems that integrate version information of different types of components in a hardware software co-design process. We contribute a novel component-oriented version management mechanism/infrastructure that is capable of capturing and versioning the logical and structural contents of components. The mechanism is based on a powerful *graph-based* versioning framework that supports any complex, nested, structured components of a system. The component dependencies are also managed.

References

[1] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Domer, and Daniel Gajski. System-on-Chip Environment (SCE): Tutorial. Technical Report CECS-TR-03-41, UC-Irvine, 2003.

[2] Daniel Gajski, Jianwen Zhu, Rainer Domer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[3] Randy H. Katz. Towards a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Survey*, 22(4):375–408, 1990.

[4] D. Leblang. The CM challenge: Configuration management that works. *Configuration Management*, 2, 1994.

[5] Tom Morse. CVS. *Linux Journal*, 1996(21es):3, 1996.

[6] Tien N. Nguyen, Ethan Munson, John Boyland, and Cheng Thao. An Infrastructure for Development of Multi-level, Object-Oriented Configuration Management Services. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 215–224. ACM Press, 2005.

[7] Rational Software. http://www.rational.com/.

[8] Xilinx Platform Studio. http://www.xilinx.com/.