Component-based Version Management for Embedded Computing System Design

Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
Ames, IA 50011, USA
tien@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. The key limitations of existing version control tools used for a hardware software co-design process include their inadequacy in representing semantics of design models and inability to manage versions of both hardware designs and associated software components in a cohesive manner. Thus, it is difficult to track the logical interdependencies between the changes to hardware and software components in an embedded computing system over time.

This paper presents an application of a well-known software engineering approach to the management of embedded systems design artifacts. Our novel *component-based version management* mechanism is capable of capturing and versioning the underlying logical contents of components in system design models and their associated software artifacts in a cohesive manner. This paper also illustrates our approach in creating a versioning system, named *EmVC*, for a hardware software co-design process.

Categories and Subject Descriptors

J.6 [**Computer Applications**]: Computer-aided design (CAD); D.2.9 [**Software Engineering**]: Management

General Terms

Management

Keywords

Version Management, 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, monitoring of environmental changes, non-destructive fault detecting, and security-enforced networking.

SAC'07 March 11-15, 2007, Seoul, Korea

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 have to produce a wide variety of artifacts that are constantly in a state of evolution: user requirements are clarified; the specification model for system level design is revised; high-level architectural designs evolve; low-level hardware component designs are changed; and software components are changed. One vital task is to manage the evolution of this complex collection of interdependent artifacts so that the information can be quickly accessed and consistently organized.

However, one of the biggest barriers is the poor interoperability between specialized version control tools for different types of artifacts at different levels of abstraction. 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 [21]. When a new version of a specification is needed, an entire new document file is created. The only way that engineers know about a new version is to look at the file's name or properties such as date of modification or timestamps.

For high-level architectural system design, they use specialized environments such as Rational for UML diagrams [24], SCE [1] for SpecC development methodology [10], or SyCE [6] for SystemC design language. Version control capabilities of these environments vary a lot. IBM Rational uses ClearCase [18] as its main configuration management system (CM). However, ClearCase handles diagrams at the text level, disregarding the logical contents of design diagrams and specifications. In contrast, 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) [32]. Software programming, hardware coding, 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 source code versioning tools such as CVS [22] or Subversion [26]. Similar to other file-oriented CM tools, they have no knowledge about the underlying semantics of software components. At the board design level, layout designs are maintained within CAD layout and routing tools with little version management support.

The poor interoperability hinders the development by making it difficult to determine whether artifacts semantically conform to each other when they evolve. Modifications to artifacts are logically related to each other. Tracking of the interdependencies be-

tween changes would greatly benefit developers in maintaining the consistency between versions of hardware and software artifacts.

To improve the interoperability, *unified* versioning tools/methods that are capable of handling *both* hardware designs and software artifacts are required. Unfortunately, version management techniques, tools, and methods used for each type of artifacts (hardware designs and software components) *cannot* well carry over the other [8, 9]. For instance, existing *software* configuration management (SCM) systems do not handle well designs of *hardware* components. With text file-oriented SCM models, the only thing known about the similarities and differences between versions of design objects or diagrams is that their textual representation files share a certain number of lines! It is largely up to developers to derive the actual changes at the diagram level from those textual differences. In the cases of parallel changes to the same file, SCM systems provide the mechanism to branch and merge changes made to unrelated lines of text. However, if textual merging is applied to SpecC or UML design specifications, we might have a semantically inconsistent resulting file.

In contrast, existing versioning approaches used in hardware design environments are not well-suited for software components, source code, or documentation. Different versions of a design of a hardware component are stored as *individual* entities in a database, and version-related relations among components (e.g. predecessor-successor and alternate-of) are represented as regular relationships between component versions as in an object-relationship model. Those approaches do not have an explicit storage representation for internal *changes* between different versions of a component. The changes between different versions must be computed by comparing different object states. Thus, change management in those approaches is less efficient if applied to source code, documentation, or software components.

While efforts to integrate versioning tools for hardware and software components have had limited success [3], the main reason for the poor interoperability problem is the lack of a unified CM model that is able to support both hardware and software design and implementation. The limitations of existing version control models include their inadequacy in representing semantics of design models and inability to manage versions of both hardware designs and associated software components in a tightly connected manner.

2 Component-based Approach

To address this version and configuration management problem in a hardware software co-design and maintenance process, we introduce a unified, component-based 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 and software artifacts.

Our version management mechanism satisfies the following requirements. Firstly, it is able to provide version control for any *structured* components and the relationships among them because components can be composite. Secondly, the mechanism has an explicit storage representation for changes between different component versions. Each version is not stored as a whole entity as in existing versioning tools for hardware designs [9, 14, 30]. Thirdly, the differences between versions are directly accessible in the repository, rather than to be computed using complex differencing algorithms. Fourthly, the mechanism is able to support document-centric artifacts such as programs and documentation files. Finally, the states of a component including structure, content, and properties at different versions are easily constructed.

This paper contributes a novel component-based CM mechanism with a *graph-based* version control framework that supports any complex, nested, structured component of an embedded system. With graph-based versioning scheme taking advantage of the Molhado versioned data model and its associated repository [23], the mechanism can support a wide range of hardware design and software components produced during a hardware software co-design process. Another significant contribution is *EmVC*, a novel CM system for hardware software co-design process for embedded systems with SpecC development methodology and field programmable gate array (FPGA) technology [10]. 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. All changes are integrally captured and tightly related to each other in a cohesive manner.

Section 3 describes our graph-based representation model. Section 4 explains how we provides fine-grained version control for components. Section 5 presents our tool, EmVC, and explains how hardware and software components are managed. Related work is discussed in Section 6 and conclusions appear last.

3 Graph-based Representation

Instead of using ASCII texts as the representation for design artifacts and associated software components, EmVC uses a graph-based representation. Graphs are commonly known, well understood, have an established mathematical basis (i.e. graph theory), and encompass a huge number of concepts, methods and algorithms [20]. This makes them very interesting from a formal as well as a practical point of view. Nodes can represent components and edges can represent all kinds of relationships between components. We use a special type of graph, called *attributed, typed, nested*, and *directed graphs* to represent composite components. This type of graph has a finite set of nodes, a finite set of edges, and two functions: $source$ and $sink$ assigning exactly one source and target node to each edge. We allow multi-graphs where different edges can have exactly the same source and sink nodes. However, we do not allow hyper-graphs, which contain hyper-edges that have more than one source and target node.

A node in our model has a unique identifier. A node does not have any value data. However, each node in a directed graph can be associated with multiple attribute-value pairs. An attribute name can be any string value and must be uniquely identified. The domain of attribute values can be any data type, possibly the *reference* type. This typed attributes accommodate multiple properties associated with nodes. Each edge in this type of graph can also be associated with attribute-value pairs in the same manner as a node. This association for edges encompasses the type of edges that have associated *edge labels*.

We also allow a directed graph to be nested within another in order to support composition and aggregation among components. Nesting is a natural way for humans to control the complexity of a system. In a nested graph, the overall complexity is reduced by allowing nodes to contain entire graphs themselves. Nested graphs are also referred to as *hierarchical graphs* [19]. The nesting structure must be acyclic. This constraint is needed to ensure that we have a proper composition mechanism, i.e., a node cannot be contained within itself. Using relation notation, $(n, m) \in nested$ denotes that n is directly nested in m.

From the practical point of view, directed graphs are often used

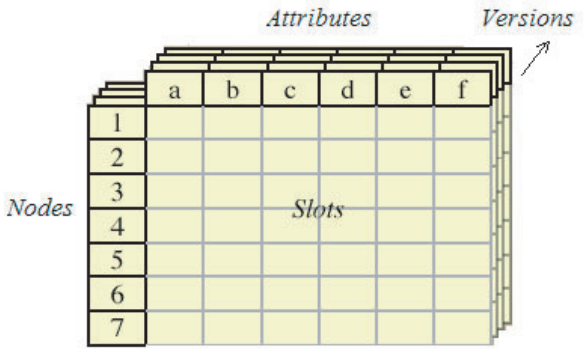


Figure 1: Data Model

as an underlying representation of arbitrarily complex software artifacts and their interrelationships in traditional software engineering environments [19]. In other words, attributed, directed graphs are sufficiently general to be used for a wide variety of components, depending on the interpretation given to nodes and edges. The nesting mechanism is attached to the graphs to facilitate the composition and aggregation among components. The nested graphs also enable an encapsulation and layering mechanism to reduce the complexity and to hide unimportant details of an artifact from others. It is apparent that many forms of nesting occur in artifacts produced during embedded system development. For example, in a hardware design model, design schemas contain composite entities. In software development, we can easily find nested methods, composite classes, packages, modules, sub-modules, etc.

Furthermore, the popular Document Object Model [5] for XML structured documents can be encoded via this attributed graph-based representation since their trees form a sub-class of this type of graph. Thus, many types of *document-centric* software artifacts that can be represented in XML-compatible formats can be accommodated via this graph-based representation.

4 Fine-grained Version Control

The previous section presented our graph-based representation model for hardware design and software artifacts. Our goal is to provide structure-oriented version control supports for all types of components in a hardware software co-design process. Therefore, a fine-grained and structure-oriented version control scheme for that type of directed graphs is required. This section presents such a scheme.

4.1 Versioned Data Model

First of all, we would like to summarize our data model. Conceptually, there are three main concepts: *node*, *slot*, and *attribute* (see Figure 1).

- *Node* is the basic unit of identity. A node has no values of its own – it has only its unique identity. The unique identifier of a node is immutable.
- A *slot* is a memory location that can store a value of any primitive data type, possibly a *reference* to a node or a set of slots. A slot can exist in isolation and can be versioned. A slot may also exist in a *container*, an entity with identity and ordered slots. A container may be heterogeneous (a *record* in which each slot has its own type) or homogeneous (a *sequence* of slots of the same type). A sequence has identity and may be fixed or variable in size. However, typically, a slot is attached to nodes, using an *attribute*.
- An *attribute* is a mapping from nodes to slots. An attribute may have particular slots for some nodes and map all other nodes to a default slot. The slots of an attribute hold values of the same data type.

In general, nodes, slots, and attributes that are related to each other form *attribute tables* whose rows correspond to nodes and columns correspond to attributes. The cells of the attribute tables are slots. Nodes can be used to represent components. The unique identifiers of nodes facilitate the history management of components, especially when they are moved around in their compositional hierarchies. Attributes and slots are used to represent object properties.

Version control is added into the data model by a third dimension in attribute tables. Technically, now there are three kinds of slots. A *constant slot* is immutable; such a slot can only be given a value once, when it is defined. A *simple slot* may be assigned even after it has been defined. The third kind of slot is the *versioned slot*, which may have different values in different versions (*slot revisions*).

The version model used in our framework (i.e. the *version* dimension) is Molhado *product versioning model* [23]. Molhado [23] is a reusable and pluggable SCM infrastructure that was designed for the rapid development of version management systems. In Molhado, a version is *global* across entire product and is a point in a *tree-structured discrete time* abstraction. That is, the third dimension in the attribute tables is tree-structured and versions move discretely from one point to another. The state of an *entire product* (including its components) is captured at certain discrete time points and only these captured versions can be retrieved in later sessions. The version model is state-based, where each version is a first class entity that represents a state of the product. A version can be associated with a name and meta-information such as date, time of modification, authors, etc. The *current version* is the version designating the current state of the entire system. Any version may be made current. When a version is set to be current, the state of the whole system is set back to that version. Every time a versioned slot is assigned a (different) value, a new version is created, branching off the current version. Molhado uses techniques derived from the work of Driscoll [7] *et al.* to store and retrieve the versioned data in different primitive data types. No file versioning is involved.

4.2 Graph-based Versioning

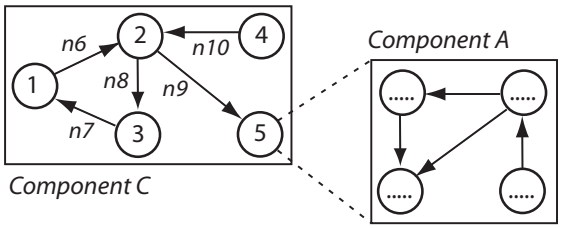
This section describes the mechanism in EmVC for storing, retrieving, and managing different versions of a software or hardware design component.

To support composite components, we have built a fine-grained version control mechanism for attributed directed graphs. An attributed, directed graph is *flatten* out as follows. An attribute table is constructed to represent an attributed directed graph. The attribute table representing for the entire graph has its rows corresponding to nodes and its columns corresponding to attributes.

(a) Each graph node is represented by a node in the table. The associated attribute-value pairs of a graph node could be easily mapped into a row of the table: attribute values are realized as slots associated with the corresponding node. The attributes in those attribute-value pairs are added into the attribute set of that table.

(b) Each edge in the graph is also represented by a new node (i.e. a new row) in the attribute table. The associated attribute-value pairs of the edge are integrated into the attribute table as in (a). Also, for the new node representing for the edge, two additional attributes are defined: "sink" attribute defines the target node of the edge, and "source" attribute defines the source node of the edge.

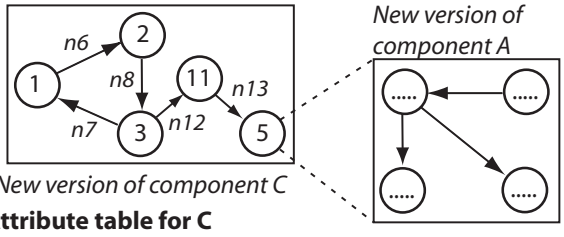
(c) For each node representing a graph node, an additional "children" attribute contains references to outgoing edges of the node.



Attribute table for C

node	"type"	"source"	"sink"	"children"	"ref"	"attr1"	....
n1	*node*	*undef*	*undef*	[n6]	null	....	
n2	*node*	*undef*	*undef*	[n8,n9]	null	....	
n3	*node*	*undef*	*undef*	[n7]	null	....	
n4	*node*	*undef*	*undef*	[n10]	null	....	
n5	*node*	*undef*	*undef*	null	comp_A	....	
n6	*edge*	n1	n2	*undef*	null	....	
n7	*edge*	n3	n1	*undef*	null	....	
n8	*edge*	n2	n3	*undef*	null	....	
n9	*edge*	n2	n5	*undef*	null	....	
n10	*edge*	n4	n2	*undef*	null	....	

Figure 2: Composite Component Representation



Attribute table for C

node	"type"	"source"	"sink"	"children"	"ref"	"attr1"	....
n1	*node*	*undef*	*undef*	[n6]	null	....	
n2	*node*	*undef*	*undef*	[n8]	null	....	
n3	*node*	*undef*	*undef*	[n7, n12]	null	....	
n4	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	
n5	*node*	*undef*	*undef*	null	comp_A	....	
n6	*edge*	n1	n2	*undef*	null	....	
n7	*edge*	n3	n1	*undef*	null	....	
n8	*edge*	n2	n3	*undef*	null	....	
n9	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	
n10	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	
n11	*node*	*undef*	*undef*	[n13]	null	....	
n12	*edge*	n3	n11	*undef*	null	....	
n13	*edge*	n11	n5	*undef*	null	....	

Figure 3: Graph-based Version Control

In our framework, a directed graph that contains other graphs will have at least one node that *logically* contains another directed graph. Let us call that type of directed graph "composite" graph and that type of node "composite" node. As described, a directed graph is used as the internal structure of a composite component. To represent the composition, for a "composite" graph, an additional attribute, attribute "ref", is created to define for each "composite" node a slot containing a reference to the component that corresponds to the subgraph nested at that "composite" node.

Figure 2 shows an example of the representation of an attributed, typed, nested, and directed graph using our versioned data model. There are two graphs in the figure: the directed graph corresponding to the component A is nested within the directed graph corresponding to the component C via node 5. The attribute table representing for component C is shown. Nodes "n1" to "n5" are "node" nodes (i.e. representing for a graph node) while nodes "n6" to "n10" are "edge" nodes (i.e. representing for an edge). Each "edge" node has "source" and "sink" slots. For example, "edge" node "n6" "connects" nodes "n1" and "n2". Each "node" node has a children slot. For example, "n2" has two outgoing edges ("n8" and "n9"). Node 5 has no outgoing edge, thus, the "children" slot of "n5" contains *null*. However, it is also a *composite* node, therefore, the "ref" slot of node 5 refers to the component A. Note that a graph node inside A might have its "ref" slot referring to another atomic or composite component. The attribute table for component A is similar (not shown).

In our graph-based versioning framework, our library functions for graphs and slots are called for the modification of the components' structures and properties. Those functions update the values of slots in attribute tables including *structural slots* (i.e. "children","parent", "source", and "sink" slots).

Figure 3 displays a new version of C and A shown in Figure 2. In the new version, the attribute table was updated by our library functions to reflect the changes to the graph structure as well as to the slot values. For example, since node 4 and edges corresponding to "n9" and "n10" were removed, any request to attribute values associated with those nodes will result in an undefined value. On the other hand, node 11 and two edges were inserted, thus, one new "node" node ("n11") and two new "edge" nodes ("n12" and "n13") were added into the table. Attribute values of these nodes were updated to reflect new connections. Attribute values of existing nodes were also modified. For example, "children" slot of "n3" now contains an additional child ("n12"), since that new edge ("n12") comes out of "n3". The attribute table for component A was similarly updated (not shown). Our storage mechanism handles efficiently these three-dimensional attribute tables.

This fine-grained versioning scheme is very efficient since common structures are shared among versions and all information including structures and contents are versioned via one mechanism. Importantly, the scheme is general for any subgraph at a node. Thus, fine-grained version control is achieved for any component that is represented by a node.

5 Tool Development

Based on the aforementioned framework and models, we have built an CM system, *EmVC*, for hardware software co-design process for embedded systems with SpecC development methodology [10]. This section describes EmVC's representation for different types of SpecC components as well as for associated software artifacts. Although no analysis tools for SpecC programs are currently provided, EmVC is used to illustrate our approach to provide versioning supports for embedded systems design artifacts.

5.1 Hardware Design Components

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* [10]. 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

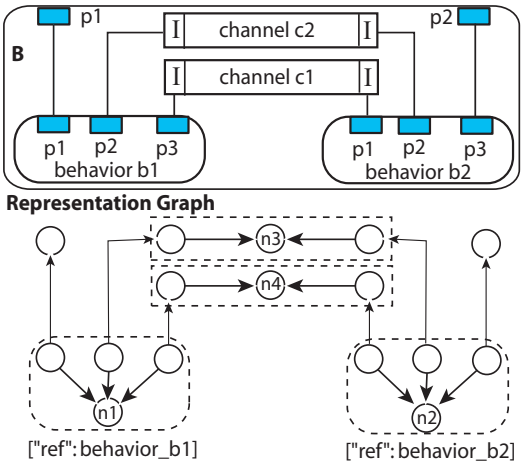
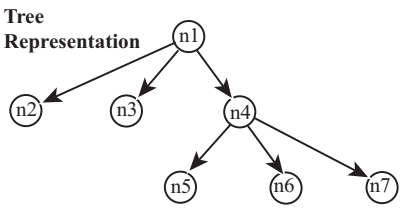


Figure 4: A SpecC "Behavior" Component



cSignal.SC

```
#include <stdio.h>
#include <stdlib.h>
channel cSignal() {
  int value;
  event ev;
  void assign(int v)
    ...
}
```

Attribute Table

	"NodeType"	"Name"	"parent"	"children"	"Parameters" ...
n1	CompiUnit	"cSignal"	null	[n2,n3,n4]	undef
n2	IncludeDecl	"<stdio.h>"	n1	null	undef
n3	IncludeDecl	"<stdlib.h>"	n1	null	undef
n4	ChanDecl	"cSignal"	n1	[n5,n6,n7]	null
n5	FieldDecl	"value"	n4	null	undef
n6	EventDecl	"ev"	n4	null	undef
n7	MethodDecl	"assign"	n4	null	[(int, v)]
...					

Figure 5: Representation for a SpecC program

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. Versioned slots are used to model other properties of a behavior. An atomic component is created to model a port, which belongs to one behavior. Port type is handled by a versioned slot.

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. Similar to behaviors, a channel is encoded by a component. 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. Similar to a port, an interface is modeled by an atomic component. Figure 4 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 internal structure is modeled as a directed graph. The "ref" attribute associated with each node contains a reference to the corresponding entities. E.g., for $n1$, the "ref" slot refers to the component "behavior_b1". The behavior B itself can be involved in other composition hierarchies.

5.2 Software Components

In this sub-section, we explain our representation for source code and document-centric artifacts. In EmVC, programs are modeled as abstract syntax trees (ASTs), which are represented via our graph representation. Similarly, documentation can be considered as tree-structured documents as in XML or HTML files.

To provide the *fine-grained* version control for code, EmVC captures the semantics of a program with a component type named *CompilationUnit*, which has a *tree-based* structure representing the program's AST. An AST node is represented as a node in the data model. Remind that to model the parent node and children nodes of an AST node, each node is associated with two attributes: "parent" attribute defines for each node the *parent* node in the AST, and "children" attribute defines a sequence of references to its children nodes. In addition to those structural attributes, each node also has an attribute ("NodeType" attribute) that identifies the syntactical unit represented by that node. Figure 5 shows an example of piece of SpecC code. EmVC's parser is able to import SpecC text files into its versioning repository's representation.

A method is represented by a node associated with a "NodeType" slot containing a method declaration, "MethodDecl". The "NodeType" slots are enumeration values of predefined AST node types. Furthermore, depending on the type of the AST node, the corresponding node has additional attributes modeling different semantic properties of the AST node. For example, for a "MethodDecl" node, the following attributes are needed: "RetType" (the return type of the method), "Name" (the method's name), "Modifier" (a string of modifiers of the method), "Parameters" (the method's parameter list), and "SourceCode". In Figure 5, not all attributes are shown. If an attribute is not applicable to a node, an *undef* value is used for the corresponding slot. If a node does not have a child, its children slot contains a *null* value. The document tree for an XML document is represented in a similar manner.

To represent the design of an embedded system, we introduce the "product" entity. "Component" entity represents a hardware or software component that belongs to a "product". Technically, a "product" is a named entity that represents the *overall logical structure* of a product. In our framework, a "product" contains a structure that is composed of components. The structure in a "product" represents the overall architectural structure of an embedded system. That structure is also implemented in the form of an attributed, directed graph or an attributed tree as in a composite component. The representation of a product is similar to a composite component. However, the "product" can *not* be used as a composite component in any compositional hierarchy. It must be regarded as the outermost composite components and sets up the context for the global version space of the product versioning model. That is, the scope of the current version is at the product level.

5.3 Logical Relation Management

Components in design models are logically related to each other and to software components. They also have interdependencies and connections with documentation and specifications. For example, an item in a specification *motivates* a high-level design of a sub-system. A SpecC channel *corresponds* to a set of variables in a SpecC program. Maintaining these logical connections among them during the hardware software co-design process is crucial for developers in understanding the system's evolution. To manage logical relations, we use hypermedia structures in which a link is represented as a *first-class* entity such as in XLink standard [33].

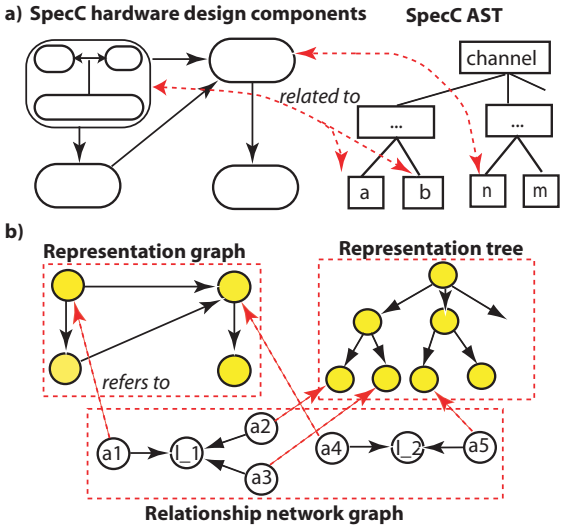


Figure 6: Logical Relations

The advantages of first-class hyperlinks have been acknowledged by hypermedia research communities [33]. For example, they facilitate the process of browsing, visualizing, and analyzing of relationship networks among components in logical models.

We have built a hypermedia model that supports first-class hypermedia structures. Let us summarize its core concepts. In that model, a *hypertext network* represents a network of relationships. A hypertext network contains *links* and *anchors*. A link is n-ary and connects a set of its anchors together. An anchor can belong to multiple links. A link or an anchor can also belong to multiple hypertext networks. An anchor does not belong to a component as in HTML. It *refers* to a graph node within a component or to entire component. To apply our versioning services to first-class hypermedia structures, we also realized our hypermedia model via the graph-based representation. In particular, a hypertext network is implemented as a component, whose internal structure is a directed graph. Each link or anchor is represented by a node in that graph. A directed edge connects an anchor's node to a link's node if the link contains the anchor. Attribute "ref" now is used to associate a slot to each anchor's node in the graph. The slot holds a reference to either a component or a graph node within a component. That node (or that component) is considered to be the position of the anchor. The use of anchors creates the separation between relationship networks and component contents. In brief, a logical relationship network is modeled and versioned according to the directed graph based versioning scheme described earlier.

Figure 6 shows our representation using first-class hypermedia structures for logical relationships between SpecC hardware design components (e.g. channels, behaviors) and SpecC program entities such as (variables, methods). Note that the logical relationships are separated from the graph and tree representing for hardware designs and software components. Changes over time to those relationships are recorded according to the graph-based versioning scheme described earlier. Details of *consistency checking* between components are beyond the scope of this paper.

5.4 Differences between Versions

One of basic functionalities in an CM system is a differencing tool which displays changes between different versions of an artifact. There are several characteristics of our framework that make our structural differencing tool simple, efficient, and accurate. The first one is the *unique* identifiers of nodes and edges in directed graphs. Second, the unique identifiers for nodes/edges are *immutable*. Third, the actual development history is accessible since the graph and attribute library functions will update values of slots whenever hardware design objects or software documentation are modified in the editors. Finally, we use the editors in which the operations will *preserve* the identifiers of nodes/edges. Therefore, changes that were actually performed from one version to another could be easily reconstructed by pairwise comparisons of versions without dealing with sequences of actual operations explicitly.

Figure 7 shows structural and logical changes between two versions $v1$ and $v4$ of a design for a FIFO Keypad Scanner. Icons that are attached to graphical objects signify the nature of the changes. For example, between $v1$ and $v4$, behavior "Decoder" was deleted ("x" icon), behavior "Keypad" was inserted ("i" icon), and properties of "Hex Keypad Code Generator" were modified (a pencil icon). With the structural difference tool for versions (see Figure 7), users are able to see changes at the *logical* level, rather than at the *textual* level as in conventional file-based versioning systems such as CVS [22] or ClearCase [18]. Furthermore, the change management in EmVC is more efficient in text-based SCM systems since logical and structural changes are directly stored in the repository, rather than being computed from *textual* changes.

Beside the structural differencing tool, another unique feature of EmVC is the ability to manage versions of hardware designs, associated software artifacts, documentation, and relationship networks among them. Figure 8 displays a snapshot from EmVC, showing a requirement specification, hardware design, and programs in SpecC and C. The hyperlinks allow users to navigate among components in different types of artifacts. Changes to hyperlink structures are also recorded.

Furthermore, all artifacts including designs, source code, and documentation are versioned in a *fine-grained* manner, thus, facilitating the semantic checking among artifacts. With text-based versioning systems, the semantic checking process is less efficient since it has to analyze the files' contents. In EmVC, versions of logical components are directly captured in the repository.

6 Related Work

Researchers in the SCM area have acknowledged the importance of version control at the component level, especially for architecture-based software development [31] and component-based software development [28]. In *SCM-supported software architecture* approach, design tools are aware of, depend on, and take advantages of SCM systems. Ragnarok [2] manages architectural evolution of a software system via a *total versioning* model. Ragnarok hides the concrete level of actual file versioning supported by CVS [22], allowing designers to work at the architecture level. SubCMTool [29] provides version control for software sub-systems but lacks of supports for connectors and interfaces. The *architecture-centered SCM* approach combines architectural and SCM concepts into a single, unified system model, called architectural system model. Using this approach, Mae [27] provides concepts of revisions, variants, optionality, and inheritance. Menage [11] manages product line architectures in terms of both time and space. To address space variabilities, Menage supports the specification of all three kinds of variation points defined by xADL 2.0 [4].

There are several approaches to managing software components in component-based software development. Koala [28] supports variability and optionality via a property mechanism. Koala does not integrate versioning information into its representation. It utilizes an external SCM system instead. Variability Categorization and Classification Model (VCCM) [13] supports the variability in

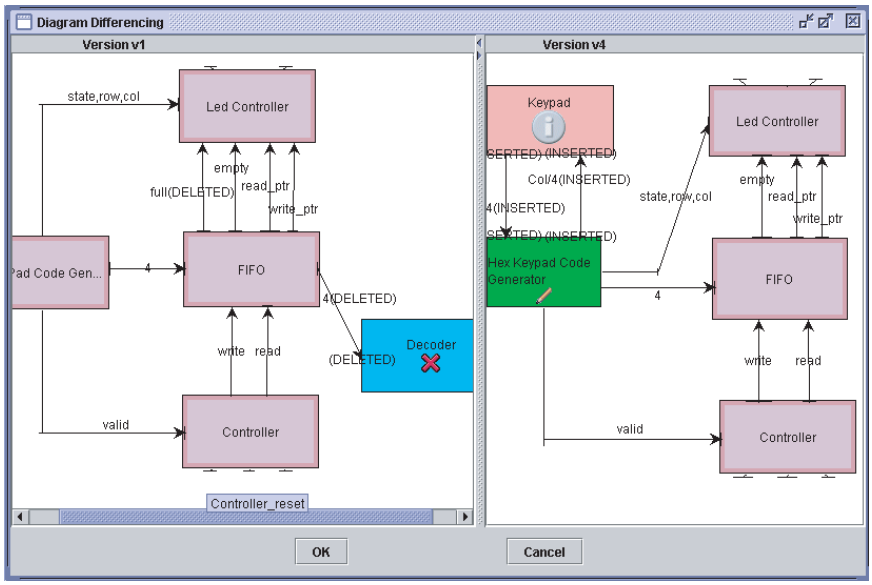


Figure 7: Differences between Versions of System Design

the software lifecycle and is used in a large-scale software product family of MRI scanners at Philips Medical Systems. Crnkovic *et al* [16, 17] introduced an SCM approach for components and relationships among them using dependency graphs. The dependency graphs are used to facilitate maintenance by identifying differences between configurations. Unicon [25] focuses on implementation-level variability. Based on a property selection mechanism, each component in a given architectural configuration is instantiated with a particular variant implementation. However, it does not capture architectural revisions and options. ShapeTool [15] does not provide mechanisms beyond grouping, versioning, and version selection. SOFA/DCUP [12] proposes a version model for components employing user-defined attribute taxonomies and entity relations.

Although those component-based version and configuration management systems have interesting functionality, they are still limited to support only a fixed set of components for a particular domain. The key departure point of our approach from existing ones is the ability to provide a unified CM model that is capable of supporting both hardware designs and software artifacts produced during the design of an embedded computing system. Also, no existing tool for embedded systems designs can support version control in the presence of hyperlinks connecting components together.

7 Conclusions

Nowadays, the development process of an embedded computing system is more and more complex, and involves concurrent hardware, software development. This places a new demand on an CM approach that can support both hardware designs and associated software components in a cohesive manner. This paper presents a novel integrated version management mechanism and its application to build an CM system, EmVC, that is capable of capturing and versioning the underlying logical contents of components in a hardware software co-design process. The mechanism is based on a graph-based versioning framework that supports any complex, nested, structured components that are involved in a hardware software co-design. The dependencies, logical connections, and relationships among hardware design components and associated software artifacts are also managed. Experimental studies on performance of the tool are being conducted. More importantly, our flexible CM mechanism can be easily used for building any versioning systems for other embedded systems development methods.

8 References

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

[2] Henrik Christensen. The Ragnarok software development environment. *Nordic Journal of Computing*, 6(1), Jan 1999.

[3] Ivica Crnkovic, Ulf Asklund, and Annita Persson Dahlqvist. *Implementing and Integrating Product Data Management and Software Configuration Management*. Artech House Publishers, 2003.

[4] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 266–276. ACM Press, 2002.

[5] Document Object Model. http://www.w3.org/dom/.

[6] Rolf Drechsler, Grschwin Fey, Christian Genz, and Daniel Groe. SyCE: An Integrated Environment for System Design in SystemC. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*. IEEE Computer Society Press, 2005.

[7] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. of Computer and System Sciences*, 38(1):86–124, Feb 1989.

[8] Jad El-khoury. Model data management: towards a common solution for PDM/SCM systems. In *Proceedings of the 12th Software Config. Management Workshop*. ACM Press, 2005.

[9] Jacky Estublier, Jean-Marie Favre, and Philippe Morat. Toward SCM/PDM Integration? In *Proceedings of the 8th Software Configuration Management Workshop (SCM-8)*. Springer, 1998.

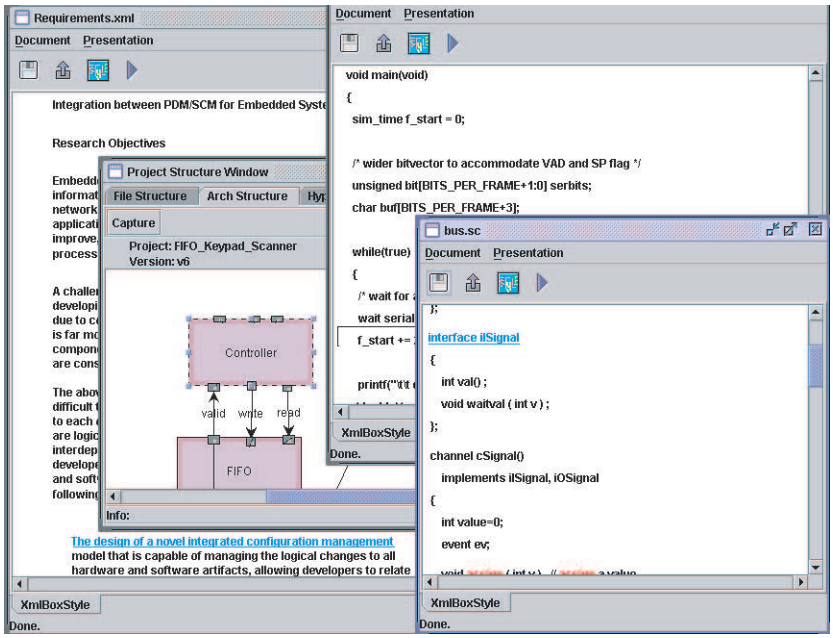


Figure 8: Hardware Software Co-design Artifacts

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

[11] Akash Garg, Matt Critchlow, Ping Chen, Christopher van der Westhuizen, and Andre van der Hoek. An environment for managing evolving product line architectures. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society, 2003.

[12] Jaroslav Gergic. Towards a versioning model for component-based software assembly. In *Proceedings of the 19th International Conference on Software Maintenance(ICSM 2003)*. IEEE Computer Society, 2003.

[13] M. Jaring, R. L. Krikhaar, and J. Bosch. Representing variability in a family of MRI scanners. *Software - Practice and Experience*, January 2004.

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

[15] J. Kuusela. Archirectural evolution. In *Proceedings of the 1999 IFIP Conference on Software Architecture*, 1999.

[16] Magnus Larsson and Ivica Crnkovic. Component configuration management. In *Proceedings of Workshop on Component Oriented Programming*, 2000.

[17] Magnus Larsson and Ivica Crnkovic. Configuration Management for Component-Based Systems. In *Proceedings of the 10th Software Configuration Management Workshop (SCM-10)*. Springer Verlag, 2001.

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

[19] Luqi. A Graph Model for Software Evolution. *IEEE Transactions on Software Engineering*, 16(8):917–927, 1990.

[20] Tom Mens. *A Formal Foundation for Object-Oriented Softw. Evolution*. PhD thesis, Vrije Universiteit Brussel, 1999.

[21] Microsoft Office. http://www.microsoft.com/.

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

[23] 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.

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

[25] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4), Apr 1995.

[26] Subversion.tigris.org. http://subversion.tigris.org/.

[27] Andre van der Hoek, Marija Mikic-Rakic, Roshanak Roshandel, and Nenad Medvidovic. Taming architectural evolution. In *Proceedings of the ACM Symposium on Foundation of Software Engineering (FSE 2001)*. ACM Press, 2001.

[28] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronic software. *IEEE Computer*, 33(3), 2000.

[29] H. Volzer, B. Atchison, P. Lindsay, A. MacDonald, and P. Strooper. A tool for subsystem configuration management. In *Proceedings of the 18th International Conference on Software Maintenance*. IEEE Computer Society, 2002.

[30] Bernhard Westfechtel and Reidar Conradi. Software Configuration Management and Engineering Data Management: Differences and Similarities. In *Proceedings of the 8th Software Configuration Management Workshop (SCM-8)*. Springer Verlag, 1998.

[31] Bernhard Westfetchtel and Reidar Conradi. Software Architecture and Software Configuration Management. In *Proceedings of the 10th Software Configuration Management Workshop*. Springer Verlag, 2001.

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

[33] W3C XML Linking. http://www.w3c.org/XML/Linking.