A Unified Model for Product Data Management and Software Configuration Management

Tien N. Nguyen, Electrical and Computer Engineering Department, Iowa State University

Abstract

Software Configuration Management (SCM) is the discipline of managing the evolution of a software system. Product Data Management (PDM) is the discipline of controlling the evolution of a product design. These two domains have been evolving independently and fairly disconnected. Nowadays, the development of modern products involves a substantial and growing part of software development. However, due to the limitations of approaches taken by both domains, efforts to build a unified configuration management (CM) model of SCM and PDM have had limited success. This paper presents a novel unified CM model and its associated CM infrastructure/tools that are configurable and tailorable to support any engineering area. Key contributions include a novel methodology, a unified CM model, and associated tools that allow for the automatic generation of CM code for supporting both hardware designs and software artifacts in multidisciplinary engineering areas.

1 Introduction

Engineering disciplines have sought to control the way physical products can be designed and realized. With software supports, *product data management* (PDM) tools are concerned with the management and change control of design and product data, that is, machine-readable data about physical objects [3, 7]. On the other hand, *software configuration management* (SCM) is the discipline of managing the evolution of large and complex *software* systems [2]. The PDM and SCM research have been evolving fairly independently. Many researchers showed that PDM and SCM have studied similar issues and their tools have much in common: design philosophies and methods, versioned data structures, etc [10, 16]. However, there exists fundamental differences.

Firstly, PDM has a well-established data representation model, called *EXPRESS* [13]. It consists of a family of object-oriented modeling languages dedicated to the description of artifacts and their constraints in a product. In contrast, SCM systems have no explicit data model or when present, their data models are weak such as ASCII texts [10]. Another significant difference is about *product model*, i.e., how a product is modeled in PDM and SCM systems. Many SCM tools consider the structure of a product as a tree of files and directories. Advanced SCM systems introduced the notion of a *system model* [17], which gives a detailed description of a software product. However, these systems (with some exceptions [9]) are based on files and/or other hard-wired concepts with predefined "horizontal" relationships among files (e.g. import dependencies in a build process). In contrast, hierarchies or "vertical" composition relationships are considered more important in PDM.

Versions and changes are also managed differently in PDM and SCM systems. In PDM, revisions of an object form a sequential series (i.e. linear versions), with no possibility of performing parallel changes. Versioning in SCM allows branching and merging of the branched tracks. Furthermore, in PDM, versioning concepts are added into a product model. That is, different versions of an object are stored as *individuals* in a database, and version-related relations among objects (e.g. predecessor-successor, option-of) are represented as relationships among object versions. PDM databases do not have an explicit storage representation for internal *changes* between object versions. In SCM, versioning is handled at the data model. Changes between file versions are managed as differences between text lines.

Nowadays, there is definitely an increasing need for a unified model between PDM and SCM that supports multidisciplinary engineering areas. However, the *integration* between PDM and SCM has had very limited success [3]. Poor interoperability occurs. Due to their differences, models, tools, and methods from one domain do not carry well over the other [8, 10]. Manual integration approach using import/export functionality causes inconsistency problems [8]. Loose integration with interoperability APIs between systems is largely ad hoc and unsatisfactory [3].

2 Unified CM Model

Unlike other approaches for PDM and SCM integration, we provide a novel methodology, tools, and infrastructure

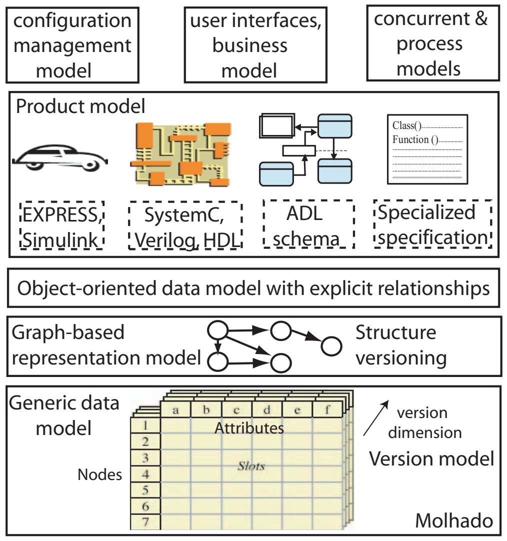
configuration management model
user interfaces, business model
concurrent & process models
Product model
Class()
Function ()
EXPRESS, Simulink
SystemC, Verilog HDL
ADL schema
Specialized specification
Object-oriented data model with explicit relationships
Graph-based representation model
Structure versioning
Generic data model
Attributes
Slots
Nodes
version dimension
Version model
Molhado

Figure 1. Unified CM Model

to build *fully* unified configuration management (CM) systems, supporting for engineering product data in multidisciplinary areas. Key ideas of our approach include modeling of products in terms of objects, fine-grained versioning of objects, and enabling version branching and structural differencing/merging of objects. The concept of *object* in our model is the same as in object-oriented models. Figure 1 summarizes the main components in our unified CM model.

Versioned Data Model: Firstly, the fundamental layer of our unified CM model is a generic versioned data model, called *Molhado* [15]. The key characteristic of Molhado is the integration of a *product versioning* model into a generic data model. Three main concepts in that data model are *node*, *slot*, and *attribute*. A node is the basic unit of identity and has no values of its own. A slot is a memory location that can store a value of any data type, possibly a reference to a node or a set of slots. A slot can exist *in isolation*. A slot may also exist in a *container*, an entity with identity and ordered slots. Typically, a slot is attached to nodes, using an *attribute*. An attribute is a mapping from nodes to slots. In general, we have *attribute tables* whose rows correspond to nodes and columns correspond to attributes. The cells of the attribute tables are slots. Version control is provided by adding a third dimension into these attribute tables. *Product versioning* model is used in which a version is *global* across entire product and is a point in a *tree-structured discrete time* abstraction [15]. That is, the third dimension in these attribute tables is tree-structured. The state of *entire product* is captured at certain discrete time points. Our persistent mechanism handles the storage of slots in versioned attribute tables. No file versioning is involved.

This integration between the data and version models allows objects' internal properties and structure to be visible to the versioning system. The main departure point of our approach from existing ones is the application of version control to an *intermediate* data model, rather than to a *high-level* product model as in PDM systems or to a *low-level* text-based data model as in SCM systems. This approach creates the flexibility for our model to support any product model involving *both* objects and document-centric artifacts. In PDM systems, the restriction to version-savvy product models (object-oriented or entity-relationship models) makes it harder to support document-centric software artifacts such as programs or documentation. In contrast, versioning at the textual data level as in SCM systems is not well-suited for structured objects.

Version Control for Objects: In our unified model, the high-level data model representing components in a product is the *object-oriented data model* with the explicit representation of relationships including structural, compositional relationships, dependencies, etc. The object-oriented model has shown its success in both PDM and software development, and is sufficiently powerful for multidisciplinary engineering areas. The chosen supporting language is Java due to its popularity and the availability of associated tools. That is, the code that will be generated from users' specifications is in term of Java classes containing CM code. Our goal is to provide CM support for components in any engineering product, which is modeled in term of objects in the object-oriented data model. Therefore, we built a mechanism to manage different versions and configurations of objects. It is built based on Molhado versioned data model with Java reflection and inheritance supports.

Two basic Java classes are provided: *"product"* and *"component"*, which represent products and their components, respectively. Representations of domain-specific products (e.g. software product, computing system) are generated as sub-classes of "product" class. Technically, a "product" is a named entity that represents the *overall logical structure* of a product. In our model, a "product" contains a structure that is composed of "components". That structure is implemented in the form of an attributed, directed graph. Depending on the application domain, the structure in a "product" will represent different forms of system structures such as the architecture of a mechanical product, the architectural structure of an embedded system, the file directory structure of a software system, etc.

Representations of domain-specific objects described in specifications are automatically generated as sub-classes of "component". It could represent a hardware design object, a sub-product, an architectural component, etc. For source code control, "component" can model program, object-oriented class, function, package, file, etc. A "component" has a unique identifier and can be versioned, saved, loaded, and exists within the version space of a product. The "component" class also sets up object loading/saving functions for derived types of objects in accordance with our persistence mechanism. The storage and retrieval of a "component" is based on the Java reflection mechanism.

In our unified model, two groups of components can be generated: *atomic* and *composite*. Composite components can share the same constituent components, and have internal structures. For an atomic or composite component, an internal property whose value changes over time is generated as a *versioned slot* with its data type. Otherwise, non-versioned slots are used. Object properties are manipulated via our APIs for slots such as "getValue" and "setValue". These functions know how to retrieve the right slot value at a version, and to modify slots at the current version.

Graph-based, Structure-oriented Versioning: The CM support for composite objects is especially important in accommodating both PDM and SCM. In our model, the internal structure of a composite object is represented by a special data structure, called *attributed, typed, nested*, and *directed graphs*. In this type of graph, each edge has exactly one source and one target node. A node or an edge has a unique identifier and can be associated with multiple attribute-value pairs. The domain of a value can be any data type. These typed attributes accommodate multiple properties associated with objects and relationships, depending on the interpretation given to nodes and edges. Our model also allows a directed graph to be nested within another in order to support composition and aggregation among objects.

A novel structure-oriented versioning algorithm for this type of graph was developed to provide the fine-grained content change and version management. The algorithm takes advantage of Molhado's storage and versioning capabilities for versioned attribute tables. That is, an attributed, typed, nested, directed graph is represented as an attribute table. Graph nodes/edges are represented as rows, and attribute values as slots in the attribute table. To support nesting graphs, our *versioned reference* mechanism is used [15]. Common structures are shared among versions and fine-grained versioning is achieved for any object that is represented by a node. Using a directed graph for object's internal structure, any structure among direct sub-components of a composite can be modeled. This is a distinguished feature from existing CM models, which support the composition of multiple sub-components but no relations between them can be defined. This type of graph also supports *document-centric* software artifacts commonly found in SCM since those artifacts can be regarded as DOM tree-structured documents [5]. Our system inserts code for manipulating a graph and slots into a generated domain-specific class.

Object-oriented CM: Configuration management is more than version control for product entities. Typical tasks include transaction services, the management of configurations, concurrent engineering, workspace, and process management. Fortunately, most of these functions can be reused from Molhado SCM infrastructure [15]. Using graph-based structure-oriented versioning and Molhado's product versioning creates a significant benefit, which is the simplicity of generated CM code to compose consistent configurations among objects. The reason is that the product versioning engine maintains the connection between versioned slots of the *same version* of a product (i.e. a configuration). So, it is able to retrieve objects in the same configuration.

Structure-oriented Differencing and Merging: Differencing and merging tools for different versions are crucial in concurrent engineering control. However, they are currently limited in PDM. In SCM, differencing/merging between file versions are handled in term of text lines. In our model, since objects' structures are represented as attributed, directed graphs, we have developed *structure-oriented* differencing and merging algorithms for that type of graph.

The structure-oriented differencing algorithm can determine if a node or an edge has been *deleted*, *inserted*, or *moved*, and if a value in an associated attribute table has been *modified*. A function to return differences between two versions of an attributed, directed graph is also provided. The algorithm is based on the following characteristics. Firstly, the *unique* and *immutable identifiers* of nodes and edges facilitate the management of object histories, especially when objects are moved. Secondly, we assume that editing environments for objects are *structure-oriented*, in which the operations *preserve* the identifiers. Also, API functions for graphs/slots are used for object manipulation. Thus, 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. On the other hand, the principle of our three-way structure-oriented merging algorithm is to analyze the presence and absence of nodes/edges and the changes of associated attribute tables in three involved versions. The algorithm is based on case scenario analysis.

Product Model: Each engineering product might have its own specification. The configurable product model in our framework (see Figure 1) is able to input specifications for different data product models (i.e. the modeling of different types of product), and *automatically* generate CM code for the corresponding types of objects and structures. Our current implementation partially supports the following types of specifications: 1) simplified EXPRESS schemas [13] for mechanical/civil engineering data objects, 2) Verilog specifications for hardware software co-design of embedded computing systems, 3) xADL schemas, XML-based architectural description language schemas [4] for *software* architectural design, and 4) XML specifications of UML diagrams. For multidisciplinary engineering products, multiple product model's specifications can be simultaneously used. Specifications can contain both object type definitions and instances of particular configurations.

Experience: Our CM infrastructure and tools are available both as an Eclipse plugin and a separate application. From the newly generated code and our CM infrastructure, the novel CM system can be built and then used in the corresponding editing/design environment. To experiment with our methods, we have built a prototype of a unified PDM/CM system for hardware software co-design with Verilog. It has illustrated the benefits of our unified model.

3 Related Work

Several sources can be served as excellent surveys on SCM [2, 3, 12] and PDM systems [1, 3]. Existing SCM systems have no or little integration and interoperation capabilities with PDM [3]. On the other hand, the awareness of software development in many PDM systems is still very low [3]. Similarities and differences between them have been systematically analyzed [10, 16]. In general, there are three possibilities for achieving the interoperability between PDM and SCM: *no integration*, *loose integration*, and *full integration* (or total unification) [3]. With no integration, *manual* intervention and procedures must be carefully identified in order to enable information exchange. Inconsistencies might occur and go unnoticed due to the disconnection.

Loose integration approach is based on relatively independent tools and well-defined interfaces [3]. The crucial elements are individual APIs of PDM and SCM, and a set of *interoperability* functions and *synchronization* monitors. Crnkovic *et al* [3] have reported a case study of a loose integration of eMatrix [6] and ClearCase [11]. This approach potentially creates data inconsistency when a change in ClearCase has not yet been registered in eMatrix. Another example of a loose integration is between Metaphase [14] and ClearCase. The integration architecture is based on a data exchange architecture. The authors also conducted six other case studies from different companies [3]. Their conclusion is that integration solutions are largely ad hoc, unsatisfactory, and complex. Unnecessary complexity is often a result of the incompatibility between PDM and SCM tools. The points are concurred by Elkhoury [8].

Full integration implies the design and implementation of new unified CM systems. Crnkovic *et al* [3] suggested an ideal full integration model, which has a common repository, a common information model, a common user interface, and individual business models with different services. Estublier *et al* [10] introduced a full integration framework with common product, evolution, and process models. Unfortunately, no detailed design, implementation is provided.

4 Conclusions

Nowadays, modern product development involves several types of artifacts, placing new demands on a unified CM model of PDM and SCM. This paper shows that PDM and SCM concepts, models, techniques are mature enough to achieve a full integration model. Our solution is based on the following ideas: modeling of products in term of objects, fine-grained CM for objects, and enabling version branching and structural differencing/merging of objects.

Key contributions include a unified CM model, method, and configurable tools that take advantage of strengths from both domains and allow for the automatic generation of CM code supporting for multidisciplinary engineering areas. Although our approach requires efforts to implement a new CM system, our code generation mechanism offers big help. Future work includes a more customizable process model and a better view-based selection mechanism.

References

[1] CIMdata, Inc. http://www.cimdata.com/.

[2] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[3] I. Crnkovic, U. Asklund, and A. P. Dahlqvist. *Implementing and Integrating Product Data Management and Software Configuration Management*. Artech House Publishers, 2003.

[4] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the Int. Conference on Software Engineering (ICSE'02)*. ACM Press, 2002.

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

[6] MatrixOne Inc. http://www.matrixone.com/.

[7] K. McIntosh. *Engineering Data Management - A Guide to Successful Implementation*. McGraw-Hill, 1995.

[8] J. El-khoury. Model data management: towards a common solution for PDM/SCM systems. In *Proceedings of the 12th Software Configuration Management Workshop (SCM-12)*, pages 17-32. ACM Press, 2005.

[9] J. Estublier. A configuration manager: The Adele database of programs. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, 1985.

[10] J. Estublier, J.-M. Favre, and P. Morat. Toward SCM / PDM Integration? In *Proceedings of Software Configuration Management Workshop (SCM-8)*. Springer Verlag, 1998.

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

[12] A. Leon. *A Guide to Software Configuration Management*. Artech House Publishers, 2000.

[13] Product Data Representation & Exchange, ISO-DIS-10303.

[14] SDRC Metaphase. http://www.sdrc.com/metaphase.

[15] T. N. Nguyen, E. Munson, J. Boyland, and C. Thao. An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 215-224. ACM Press, 2005.

[16] B. Westfechtel and R. Conradi. Software Configuration Management & Engineering Data Management: Differences and Similarities. In *Proceedings of Software Configuration Management Workshop (SCM-8)*. Springer Verlag, 1998.

[17] A. Zeller. Versioning System Models through Description Logic. In *Proceedings of Software Configuration Management Workshop (SCM-8)*. Springer Verlag, 1998.