SubCM: A Tool for Improved Visibility of Software Change in an Industrial Setting

Hagen Völzer, Anthony MacDonald, Brenton Atchison, *Member*, *IEEE*, Andrew Hanlon, Peter Lindsay, and Paul Strooper, *Member*, *IEEE*

Abstract—Software Configuration Management is the discipline of managing large collections of software development artefacts from which software products are built. Software configuration management tools typically deal with artefacts at fine levels of granularity—such as individual source code files—and assist with coordination of changes to such artefacts. This paper describes a lightweight tool, designed to be used on top of a traditional file-based configuration management system. The add-on tool support enables users to flexibly define new hierarchical views of product structure, independent of the underlying artefact-repository structure. The tool extracts configuration and change data with respect to the user-defined hierarchy, leading to improved visibility of how individual subsystems have changed. The approach yields a range of new capabilities for build managers, and verification and validation teams. The paper includes a description of our experience using the tool in an organization that builds large embedded software systems.

Index Terms—Software configuration management, software maintenance, verification and validation.

1 Introduction

1.1 Motivation

Software Configuration Management (CM) [5], [13], [29], [36] is a key discipline for development and maintenance of large software systems. To motivate the need for improved hierarchical CM support, we first describe how CM is currently performed in organizations that develop multiple related software products, in different configurations for different customers [22]. Typically, there are many different types of *artefacts* to maintain (source code, binaries, documentation, etc.) and each artefact requires individual version control. Typically, a Configuration Control Board (CCB) oversees change management, project managers coordinate the change implementation process, and a software *build process* is used to integrate individual changes into *baselines* to build new product versions [27]. Product versions consist of different configurations of artefact versions, and these configuration structures evolve over time.

A wide range of CM tools are available to support such activities [7]. Traditionally, CM tools provided support primarily at two levels of granularity:

- at the level of basic artefacts, such as source-code files, documents and test sets—called *atomic Configuration Items* below, and
- at the level of system baselines, consisting of a complete snapshot of the system at key stages in its development, such as when components are released for verification and validation (V&V), or when products are released to customers.

Modern software development practices introduce hierarchical structuring of systems into *subsystems* (aggregations of software artefacts) to improve manageability. The problem of handling Configuration Items (*CIs*) at arbitrary levels of granularity has been recognized for some time [4], [36]. Most CM repositories do support some form of hierarchical structuring, but there is a tendency for system hierarchies to be put in place early and to change little thereafter. Those repositories that do support hierarchical structuring provide limited or no versioning of the intermediate levels, which means organizations need to develop their own procedures for retrieving versioning and change information of subsystems.

1.2 Contributions

This paper reports on the evaluation of an approach to *subsystem-based CM* that aims to enable CM at flexible, intermediate levels of granularity. Candidates for subsystem-based CM include:

- coarse-grained items such as product families and their major components;
- medium-grained components such as middleware, test suites and the modules implementing particular communications protocols; and
- smaller, shared components such as calculation functions in a code library and interface objects.

Fig. 1 illustrates the kinds of hierarchical structures the approach aims to support: It shows a product graph [9] for a family of product versions. In our approach, organizations are assumed to have CM support in place for system

- H. Völzer is with the University of Lübeck, Ratzeburger Allee 160 23538, Lübeck, Germany. E-mail: voelzer@tcs.uni-luebeck.de.
- B. Atchison and A. Hanlon are with the Invensys SCADA Development, PO Box 4009, Eight Mile Plains, Qld 4113, Australia. E-mail: {brenton.atchison, andrew.hanlon}@invensys.com.
- P. Lindsay, A. MacDonald, and P. Strooper are with the School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Qld 4072, Australia. E-mail: {pal, anti, pstroop}@itee.uq.edu.au.

Manuscript received 27 May 2003; revised 1 Mar. 2004; accepted 16 July 2004.
Recommended for acceptance by J. Offutt.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0061-0503.

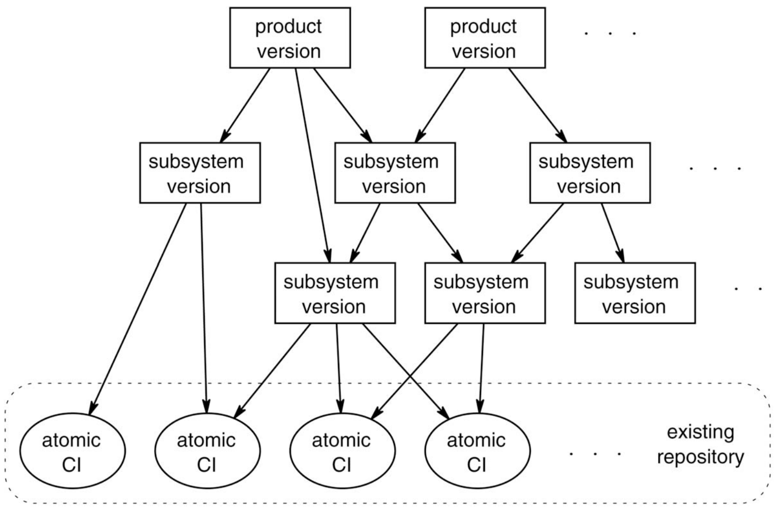


Fig. 1. Hierarchically structured configuration items.

versioning and change tracking of atomic CIs. We then provide lightweight, add-on tool support that allows users to define their own system hierarchies and provides automated support for viewing configuration and change information for the resulting subsystems, mechanically extracted from data in the existing CM repository. Subsystem hierarchies can be defined independently of the system structure on which the underlying CM repository is based.

We defined requirements for the approach in an earlier paper [19] and later showed what kind of tool support would be needed [33]. The current paper reports on the adaptation and trial of a prototype tool in a company that develops and maintains large embedded software systems. The evaluation revealed the strengths—and some weaknesses—of the approach and yielded new insights into ways in which subsystem-based CM can improve the software process.

The result of the evaluation was a subtle but profound reassessment of the goals of subsystem-based CM. Our original goal was to provide proactive, forward-looking management of subsystems, so that subsystem-based CM would be used to determine and manage enhancements to systems. As we worked with industry personnel, it became apparent that software developers would resist the introduction of an approach that interfered with their CM practices without offering them tangible benefits. Build managers and verification and validation (V&V) personnel on the other hand could see potential benefit from visibility of subsystem configuration and change data. Hence, over the duration of the project, the main goal became to provide visibility of subsystem configuration and change data to build managers and V&V personnel. We believe the revised approach offers an effective lightweight add-on to existing CM tools. Since CM repositories typically represent a critical investment for organisations, such considerations are extremely important.

1.3 Paper Structure

Section 2 outlines our approach to subsystem-based CM, which we call *SubCM*. Section 3 presents the revised conceptual model for subsystem-based CM on which our approach is based. Section 4 describes a prototype SubCM Tool developed to support the approach; the prototype interfaces with Telelogic's CMSynergy tool [28] but is general enough to interface easily with other CM tools. Section 5 is concerned with how the tool might be integrated into the software lifecycle, including the development, maintenance, and V&V processes. Section 6 reports on experience applying the SubCM Tool to products developed by the Invensys SCADA development group, who develop technology for the Supervisory Control and Data Acquisition (SCADA) domain. Section 7 describes related work and includes comparisons with related research projects and existing CM tools.

2 Outline of Approach

The SubCM approach is intended to extend the capabilities of existing CM approaches to subsystems, by addressing issues such as the following:

- Configuration information: What is, and what is contained in, a given version of a subsystem? Conversely, which subsystem versions (if any) contain a given object.
- Change data: In what ways has a subsystem (and its constituents) changed between versions? What caused a change to a subsystem, and how was the change carried out and checked?
- Variant comparisons: How do two versions of a subsystem differ?

The two key steps to using the approach are to define the subsystem—or more generally, the subsystem hierarchy—that is of interest and the system baselines across which it is to be investigated. This section overviews the two steps; Section 3 defines the underlying data structures in more detail and Section 4 describes the SubCM Tool.

In the first step of subsystem definition, the user defines a *subsystem hierarchy* by nominating a set of subsystems and their constituents. Subsystems can consist of other subsystems and atomic CIs, and constituents can be shared

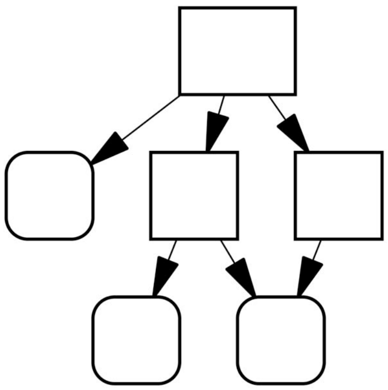


Fig. 2. Example subsystem hierarchy, with subsystems (square corners), atomic CIs (rounded corners), and references (arrows).

between subsystems. Conceptually, the hierarchy is specified as a directed acyclic graph, corresponding to a product graph [9] with atomic CIs at the leaf nodes and the arcs representing references between a subsystem and its constituents: see Fig. 2. The user needs to name the subsystems and specify the identifiers of the atomic CIs. In principle, the atomic CIs can reside across different CM repositories, for example when code is stored separately from documentation or when code is stored in multiple CM repositories: In such cases, the user needs to identify the appropriate repository and supply the identifier used by that repository. However, the SubCM Tool currently has only been prototyped with a single CM repository.

In the second step, the user nominates a collection of system baselines, called the *baselines reference basis*. Typically, this will be a branching structure with branches corresponding to the "is a modification of" relationship between different baselines of the system [34]. Fig. 3 shows the version graph [9] for an example. The user decides which baselines to include in the reference basis: For example, they might decide to include all development releases, only releases to V&V, only product releases to customers, or some arbitrary combination of these. Typically, the reference basis will contain only a subset of all possible baselines, and its members need not represent consecutive baselines.

Given a subsystem hierarchy and a baselines reference basis, the SubCM Tool inspects the CM repository and builds a *subsystems version graph* corresponding to each baseline in the reference basis: see Fig. 4, where the dashed arrows represent the derived "is a modification of" relationship between subsystems and the vertical arcs represents the relationship between the baselines and the corresponding subsystem versions. For each atomic CI in the hierarchy, if the CI was included in the baseline, then its particular version is noted in the graph for that baseline; otherwise, it is omitted. (In the example, the atomic CIs in the graph corresponding to baseline 1 happen to be labeled version 1 but no significance should be read into this.) The SubCM Tool also provides support for manually changing the structure of subsystem hierarchies, which is similar to the setup process.

The tool assigns version numbers to subsystems as follows: All subsystems in the first baseline (the root of the reference basis) are labeled as root versions. Where a subsystem has not changed in any way between two successive reference baselines, the subsystem version identifier does not change. Changes percolate up the hierarchy: If the version of one of its constituents changes, the subsystem is deemed to have changed, and a new version of the subsystem is created. Where a subsystem has changed between baselines, a version identifier is created for the new subsystem using an appropriate numbering convention. (The prototype tool uses the numbering convention from the underlying CM system.) The tool also creates a *change description* (denoted by "CD" in Fig. 4) that indicates which constituents have changed and how, using the change information of atomic CIs in the underlying CM repository. Note that subsystem version identifiers are relative in this approach: They depend on the hierarchical structure chosen (including the nesting of subsystems within subsystems) and on the baselines reference basis chosen. For absolute identification of subsystems in this approach, it is enough to know to which system baseline (or baselines) the system version corresponds.

The SubCM Tool provides functions for viewing subsystem configuration and change data in a number of different formats, including:

- what versions of what artefacts made up the subsystem at any given time (i.e., for the nominated baselines),
- what system baselines contain a particular subsystem version,
- what changes were made to a given subsystem between two baselines on the same development branch, and
- what changes were made to a given subsystem in a given release that *weren't* made to the "same"

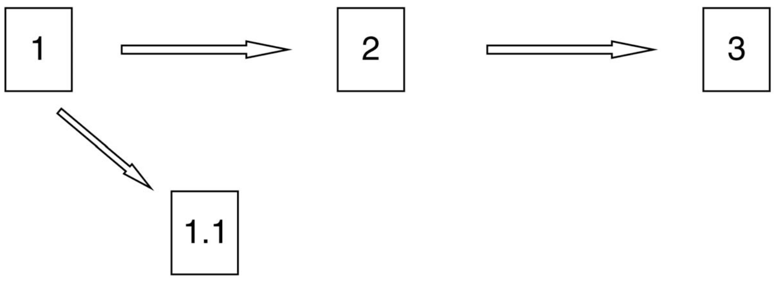


Fig. 3. Example baselines reference basis, consisting of version identifiers for user-selected system baselines; the form of branching will depend on the version branching model used by the underlying CM system [34].

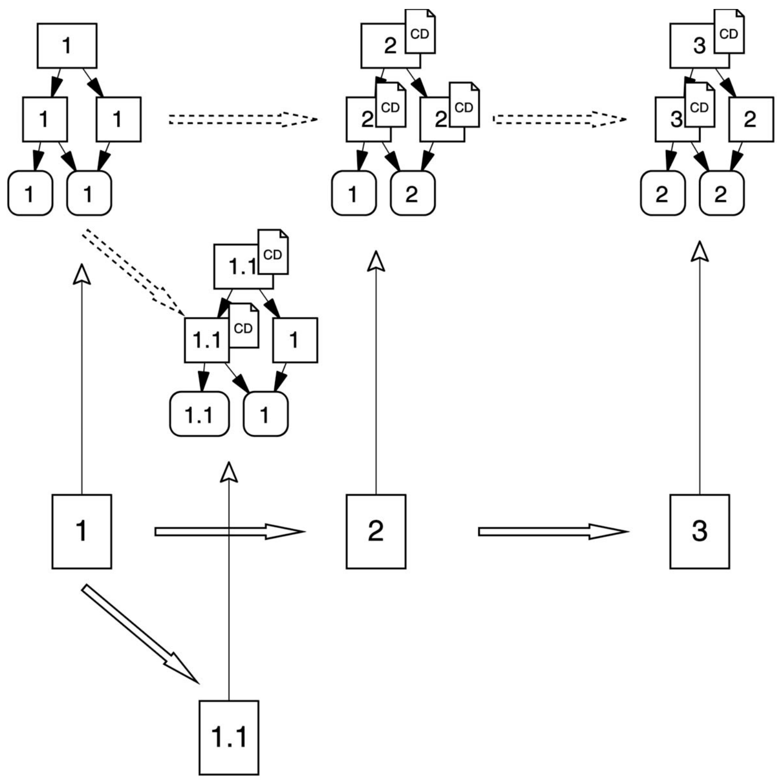


Fig. 4. Example of the subsystem version graphs generated and maintained by the SubCM Tool for a given baselines reference basis (bottom half of the figure).

subsystem in another given release, possibly on a different development branch.

The strength of the approach is that it allows subsystems to be defined retrospectively and their "histories" constructed automatically, in hindsight and in abstract (i.e., what was this subsystem's development path with respect to given system baselines).

3 The Underlying Model

This section defines the underlying model in detail and introduces the terminology used in the SubCM Tool. It is an extension of the model published earlier [19], based on implementing the SubCM Tool and its industrial trial. The main changes have been additional information to characterize subsystems and change descriptions and the additional ability to compare variants of subsystems.

A *subsystem* is a logically coherent collection of software development artefacts, such as specification documents, design documents, source code, binaries, user documents, build and testing resources, build and test reports, requirements tracing documents, and release notes. As noted in Section 1, subsystems occur at many levels of granularity.

The model supports the following three capabilities:

1. characterization of the subsystem configuration, via *Subsystem Configuration Specifications (SCSs)*,
2. characterization of the change between two consecutive versions of a subsystem, via *change descriptions*, and
3. characterization of the difference of two versions of a subsystem via a *variant comparison*.

These are explained in more detail below and illustrated on an example subsystem, the DNP protocol, that is a constituent of many products at Invensys.

3.1 Subsystem Configuration Specifications

A *Subsystem Configuration Specification (SCS)* states which versions of which objects make up a particular version of a subsystem: See Fig. 5 for an example. An SCS consists of a subsystem identifier, which contains a name and a version identifier, a textual summary of the subsystem (a description of what makes the collection logically coherent), and a set of *constituents*. A *constituent* is a reference to another subsystem or an *atomic CI*. *Atomic CI* is our term for a configuration item that is managed by external tools, such as source-code files and documents. Atomic CIs are identified by a name, a version number, and a location such as a database identifier. Finally, a *type*, such as "subsystem," "user doc," or "source code" is associated with each constituent.

3.2 Change Descriptions

In our approach a *change description* provides an abstract description of how the subsystem has changed between two versions. It consists of the two version identifiers, a textual description of the change, and a set of *change items*. A *change item* describes a change of a particular constituent. It consists of a constituent identifier, a *change type*, which describes how the constituent has changed, and a set of associated *change references*. The model considers the change types *none* (the constituent has not changed), *added* (the constituent was added between the old and the new version), *deleted* (the constituent was deleted from the old

Name: DNP
Version: 3
Description: DNP protocol for Remote Terminal Unit

Constituents	Version	Type	Location
Core	1	subsystem	–
Master	3	subsystem	–
Slave	2	subsystem	–
Makefile.mak	1	makefile	SrcCode

Fig. 5. SCS for a subsystem with three subsubsystems.

configuration), and *modified* (some content of the constituent has changed and the constituent appears in a different version in the new subsystem version). A *change reference* is a pointer to where details of the change can be found. The particular nature of change references and the traceability they provide will be specific to the underlying CM system. In the prototype described below, they refer to the identifier of the task that modified the constituent.

Fig. 6 shows a *change description* for the DNP protocol. More detail can be obtained by looking into the change descriptions of the constituents and by looking at the information that is associated with the change references. Note that, for each change description, there is an SCS describing the new configuration and for each SCS that is not a root version, there is a corresponding change description describing how it was derived from its parent version.

We also consider change descriptions for two nonadjacent versions on a linear version history. For example, a change description for version 5 with respect to version 3 can be derived by *aggregating* the change descriptions for version 5 with respect to version 4 and the change description for version 4 with respect to version 3. Two consecutive change descriptions are aggregated by concatenating their summaries and, for each constituent, aggregating the two change items for that constituent. Two change items of the same constituent are aggregated by unifying their sets of change references. For example, the change items

Master–modified (from 2 to 3)–refs 5134, 5135

and

Master–modified (from 3 to 4)–refs 7130, 7132

aggregate to

Master–modified (from 2 to 4)–refs 5134, 5135, 7130, 7132.

3.3 Variant Comparison

In the previous section, we presented the concept of an aggregated change description, which characterises the difference between two versions that are related via a chain of adjacent versions. This concept can also be used to compare parallel versions (i.e., variants) of a subsystem as follows: Two variants x and y of a subsystem are related via a youngest common ancestor version z as shown in Fig. 7. For each pair (z, x) and (z, y), there is an aggregated change description as defined in Section 3.2. The pair of these two change descriptions constitutes the *symmetric variant comparison* between x and y.

Name: DNP
Current version: 3
Parent version: 2
Summary of changes: DNP changed in this version as a consequence of changes to the Master and Slave subsystems. Master underwent trivial changes, however Slave changed significantly (see Slave change description for details).
Change Reference Summary: 1215, 1246, 1249, 1265, 1266, 1280

Item	Change type	Change References
Core	none	
Master	modified	1215
Slave	modified	1246, 1249, 1265, 1266, 1280
Makefile.mak	none	

Fig. 6. Change description for version 3 of DNP.

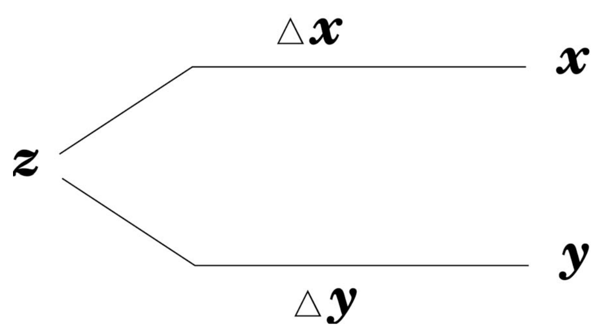


Fig. 7. Two parallel versions x and y, and their youngest common ancestor z.

With each of these two change descriptions, there is an associated set of change references: The set Δ_x of changes that affect the subsystem between the versions z and x, and the set Δ_y of changes that affect the subsystem between the version z and y. A useful characterization of the difference of the variants is the difference of the changes that are associated with each version, i.e., we are interested in the set $\Delta_x \setminus \Delta_y$ of changes that are included in version x but not in version y of the subsystem. This difference constitutes the *asymmetric variant comparison* of x with respect to y.

Fig. 8 shows an example of an asymmetric variant comparison. It shows the version identifiers of both variants, a summary of the references to the changes that are included in version 1.1.1 but not in version 3, and a list of which of these changes affect which configuration items.

4 The Tool

This section describes tool support for the above model. The approach is quite general and would work with most CM systems; we demonstrate it by describing the prototype that was trialled at Invensys.

The data associated with a subsystem version—its SCS and change description, and the system baselines with which it is associated—are stored locally by the SubCM Tool while atomic CIs are stored and managed by the underlying CM system. Subsystem version information can be exported to a data file for reporting purposes. The tool presents two navigable views onto subsystem version data: the *Configuration View* for SCS and baselines information and the *Change View* for change descriptions and for variant comparison. The different views and their navigation facilities are described in more detail below and illustrated with hypothetical examples based on Invensys's middleware.

4.1 The Underlying CM Repository

The SubCM Tool prototype interfaces to the Telelogic CM Synergy toolset [28]. CM Synergy is a commercial, task-based CM tool which works on top of a relational database. The SubCM Tool prototype is implemented in Python [23] with use of the graphical toolkit wxPython. It connects to CM Synergy via CM Synergy's command-line interface. By using this command-line interface, the SubCM Tool extracts all the necessary information about the atomic CIs, and it then stores and manages all the subsystem information itself.

CM Synergy has three levels of change management (see Fig. 9): *change requests*, *tasks*, and *atomic changes*, which are described in more detail below. Other approaches are possible and have their own terminology: For example, Eick et al. [15] use the terms *initial modification request* and *modification request* for essentially the same things as CM Synergy's change requests and tasks. We chose to use tasks as our primary means for change references since this was the practice at Invensys (and is the practice recommended by CM Synergy's distributors).

Under CM Synergy a *change request* is created when a defect is reported or the Configuration Control Board decides to add functionality or otherwise improve a product. A change request is then broken down into tasks and each task is assigned to a developer. To enact a task, the developer checks out atomic CIs, modifies them, and then checks them back into CM Synergy. (Depending on the nature of the object and the nature of the task, the developer may need to perform other activities to complete the task, such as having the change reviewed and tested.) We use the term *atomic change* for the resulting modification to an atomic CI. Each atomic change belongs to one task. The relationship between change requests and tasks, and between tasks and atomic changes is maintained and can be retrieved from CM Synergy. The different levels of change management provide different views of the change

Name: DNP
Variant x: 1.1.1
Variant y: 3
Change References Summary: 1788, 1789, 1790, 1800, 1804

Item	variant x	variant y	Change References
Core	1	1	
Master	1.1.1	3	1788, 1789, 1790
Slave	1.1.1	2	1800, 1804
Makefile.mak	1	1	

Fig. 8. Asymmetric variant comparison for version 1.1.1 with regard to version 3 of DNP.

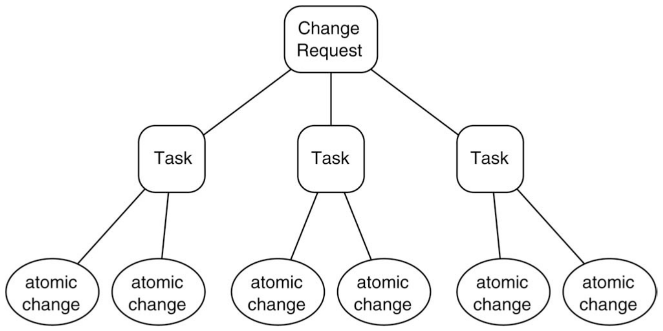


Fig. 9. Types of change.

to different stakeholders: Atomic changes and tasks constitute the view for developers, tasks and change requests provide the view for development team leaders, while change requests are the most interesting level for CCB members and customers.

4.2 The Configuration View

The Configuration View enables navigation of components of a particular subsystem version. A component can be any constituent of a subsystem, its constituents, and so on, down to and including atomic CIs. The user selects a particular component and the Configuration View will display configuration information about that component.

Fig. 10 shows a Configuration View for version 7 of the subsystem DNP3. The main window (on the left) shows the *component hierarchy window*, which consists of all the subsystem components of the root subsystem, displayed in browsable structure akin to a filesystem tree. The tree displays the name and the version of each component subsystem, in the format <name>-<version>. In Fig. 10, the selected component is DNP_Master_Files; in the subsystem in question (DNP3-7), version 4 of this component is used.

The other three windows of the Configuration View give different information about this component:

- the *description window* (upper right), which contains a textual description of the component,
- the *constituents window* (lower right), which displays version information for the constituents of the selected component, and
- the *baselines window* (middle), which displays the reference baselines in which this component and its successor versions reside.

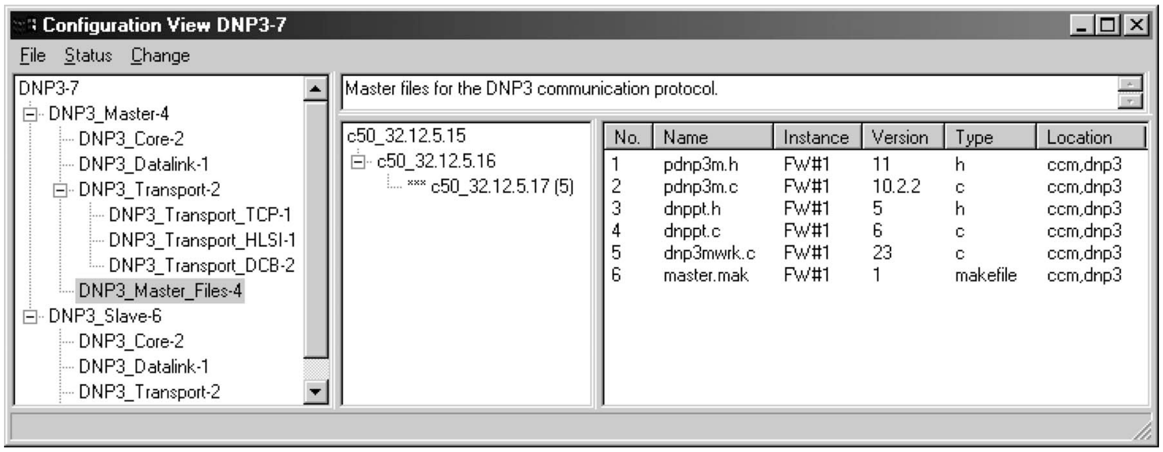
The last two of these will be explained in detail below.

The constituents window displays version information for the constituents of the selected component. Atomic CIs are identified by giving their name, instance (used by CM Synergy to distinguish between different objects with the same name) and location (indicated by the keyword *ccm* and the name of the corresponding CM Synergy database). Constituents which are subsystems are identified by name and indicated by having the keyword *scm* in the location field (indicating that their configuration data is stored by the SubCM Tool). The list of constituents in the right lower window can be sorted by name, type, or location.

The baselines window shows the tree of CM Synergy baselines that contain this particular version of the component, together with the first subsequent baselines in which the component version has changed (indicated by "***"). Thus, Fig. 10 indicates that version 4 of DNP3_Master_Files is present in baselines c50_32.12.5.15 and c50_32.12.5.16, but was replaced by version 5 in baseline c50_32.12.5.17. The number in parentheses after the baseline identifier indicates the new version of the component. The baselines window gives an impression of the stability of the selected component, in terms of how many baselines it was included in before it got changed.

The baselines window supports some basic navigation in the version history. Upon selection of a leaf of the tree, which is labeled with "***" and, hence, refers to a successor version, a new configuration view is opened with the corresponding successor version of the selected SCS. There are two more navigation functions:

- Each atomic CI can be accessed directly from the configuration view, i.e., the tool launches the appropriate viewer.
- For each constituent of the selected SCS, a list of all subsystems that use this constituent can be produced and from that list a subsystem can be selected, which



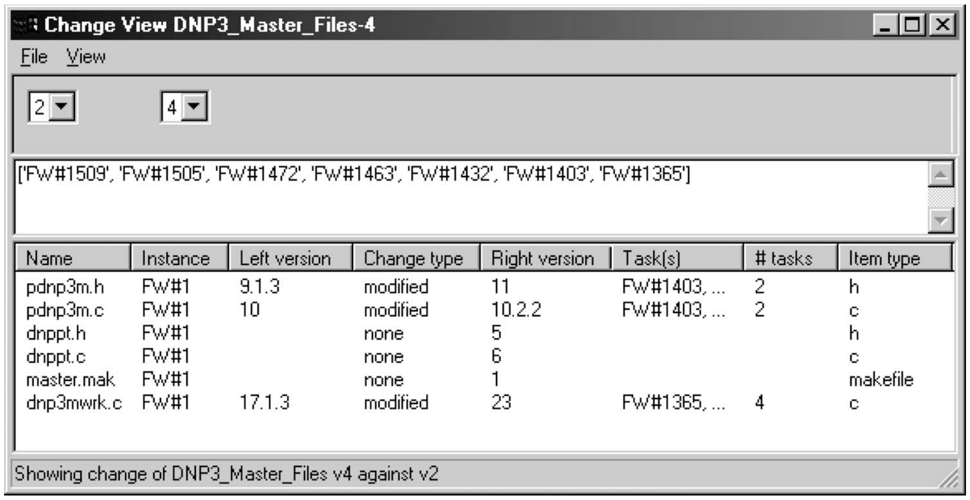
```
DNP3-7
  DNP3_Master-4
    DNP3_Core-2
    DNP3_Datalink-1
    DNP3_Transport-2
      DNP3_Transport_TCP-1
      DNP3_Transport_HLSI-1
      DNP3_Transport_DCB-2
    DNP3_Master_Files-4
  DNP3_Slave-6
    DNP3_Core-2
    DNP3_Datalink-1
    DNP3_Transport-2
```

Master files for the DNP3 communication protocol.

```
c50_32.12.5.15
  c50_32.12.5.16
    *** c50_32.12.5.17 (5)
```

No.	Name	Instance	Version	Type	Location
1	pdnp3m.h	FW#1	11	h	ccm,dnp3
2	pdnp3m.c	FW#1	10.2.2	c	ccm,dnp3
3	dnppt.h	FW#1	5	h	ccm,dnp3
4	dnppt.c	FW#1	6	c	ccm,dnp3
5	dnp3mwrk.c	FW#1	23	c	ccm,dnp3
6	master.mak	FW#1	1	makefile	ccm,dnp3

Fig. 10. A Configuration View of the DNP3 subsystem, with configuration detail of the DNP3_Master_Files component shown.



['FW#1509', 'FW#1505', 'FW#1472', 'FW#1463', 'FW#1432', 'FW#1403', 'FW#1365']

Name	Instance	Left version	Change type	Right version	Task(s)	# tasks	Item type
pdnp3m.h	FW#1	9.1.3	modified	11	FW#1403, ...	2	h
pdnp3m.c	FW#1	10	modified	10.2.2	FW#1403, ...	2	c
dnppt.h	FW#1		none	5			h
dnppt.c	FW#1		none	6			c
master.mak	FW#1		none	1			makefile
dnp3mwrk.c	FW#1	17.1.3	modified	23	FW#1365, ...	4	c

Showing change of DNP3_Master_Files v4 against v2

Fig. 11. A Change View of the DNP3_Master_Files subsystem.

will then be opened in a new configuration view. Multiple configuration views can be open at a time.

4.3 The Change View

The Change View shows change descriptions of a subsystem. In our implementation, task identifiers play the role of the change references described in Section 3.2. A task identifier allows a user to access information associated with a task, which is stored in the CM Synergy repository, such as the developer who has enacted the task, a description of which problem the task is meant to solve, etc. From the task identifier other forms of documentation can be reached, including the detailed *Program Amendment Document* that records how the item has been modified or the *System Incident Request* that details the initial cause that required change.

Fig. 11 shows a Change View for version 4 of the subsystem DNP3_Master_Files. The upper field allows the user to select two arbitrary versions of the subsystem. If the first version is an ancestor of the second, the corresponding (aggregated) change description is displayed. If the versions are variants of each other, then the change view will show the variant comparison (see Section 4.4). In Fig. 11, the change between versions 2 and 4 is shown. The upper part shows the set of tasks that changed the subsystem from version 2 to version 4 as well as the textual summary if there is any. The lower part shows how individual constituents have changed.

In addition to displaying the change description information described in Section 3.2, the Change View shows the type of each configuration item and how many different tasks were involved in the change. The list of change items can be sorted by name, change type, item type, or task. If the change description for two adjacent versions is shown, the textual summary can be edited and saved.

The Change View also supports navigation of subsystem changes. The user can select a change item that refers to a subsystem, and the tool will open a new Change View corresponding to this subsystem change. If the user selects a change item that refers to a text file (e.g., source code), the tool will open a new window that shows the output of the UNIX-diff function applied to the old and the new version of the file. Another function searches for all subsystems that include the selected change item.

4.4 Variant Comparison

The Change View can also be used to show an asymmetric variant comparison. Fig. 12 shows a variant comparison within a Change View. Compared is version 1.1.1 with respect to version 3 of a subsystem. The upper part shows the set of tasks that affect the subsystem and that are in version 1.1.1 but not in version 3 of its configuration. The lower field shows its configuration in both variants and which individual constituents are affected by each of the listed tasks.

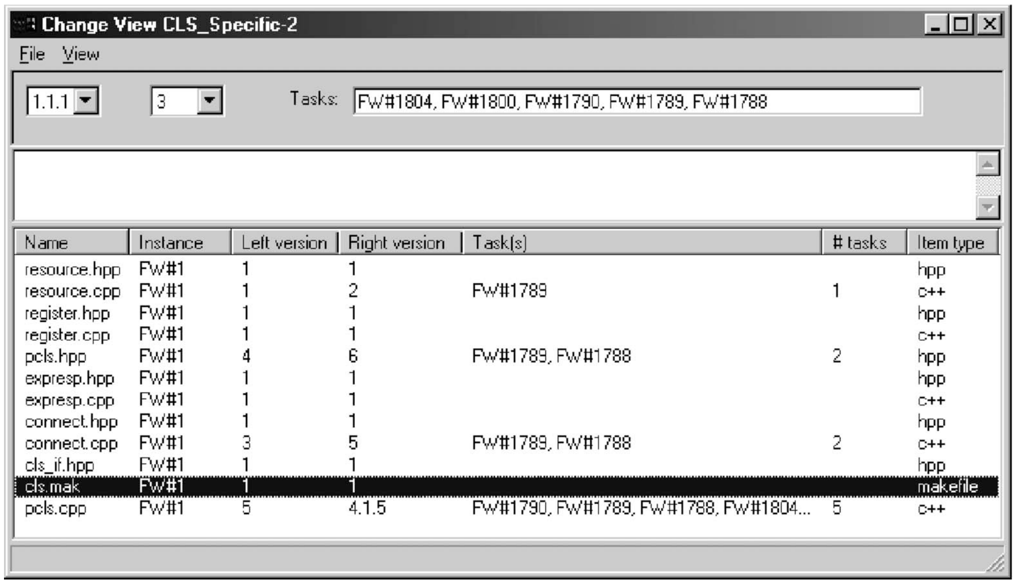
5 Tool Use

This section describes the support provided by the prototype tool for setting up a new system hierarchy and baselines reference basis and discusses ways in which the tool can be integrated into the software lifecycle.

5.1 Setting Up and Maintaining SCSs

When a subsystem configuration is defined for the first time, a root version SCS needs to be set up. (Root versions do not have parent versions, so there is no associated change description.) The creation of an SCS can be initiated from the Configuration View. The name and the summary of the subsystem have to be supplied by the user. Constituents are added one by one. Atomic CIs stored in CM Synergy can be specified by name; a list of matching object versions is then presented, from which the user selects one. In the prototype, all atomic CIs that are added to the same hierarchy must be taken from the same user-selected baseline. We say that the SCS *corresponds* to that baseline. Subsubsystems can similarly be specified by name, in which case a list of versions of the subsystem stored in the SubCM Tool is presented, from which the user selects one. Again, each subsubsystem must correspond to the same user-selected baseline.

Automated support for populating SCSs is provided with the user selecting a particular baseline and a directory



1.1.1 3 Tasks: FW#1804, FW#1800, FW#1790, FW#1789, FW#1788

Name	Instance	Left version	Right version	Task(s)	# tasks	Item type
resource.hpp	FW#1	1	1			hpp
resource.cpp	FW#1	1	2	FW#1789	1	c++
register.hpp	FW#1	1	1			hpp
register.cpp	FW#1	1	1			c++
pcls.hpp	FW#1	4	6	FW#1789, FW#1788	2	hpp
expresp.hpp	FW#1	1	1			hpp
expresp.cpp	FW#1	1	1			c++
connect.hpp	FW#1	1	1			hpp
connect.cpp	FW#1	3	5	FW#1789, FW#1788	2	c++
cls_if.hpp	FW#1	1	1			hpp
cls.mak	FW#1	1	1			makefile
pcls.cpp	FW#1	5	4.1.5	FW#1790, FW#1789, FW#1788, FW#1804...	5	c++

Fig. 12. An asymmetric variant comparison shown in the Change View.

within CM Synergy. All the objects in that directory that exist in the selected baseline are then extracted, and a reference to the object and its corresponding version is added to the SCS. This semiautomatic population of a subsystem can also be applied recursively to subdirectories.

5.2 Selecting Baselines and Generating Change Descriptions

Two ways of selecting baselines are supported: a single-step approach and a fully automatic approach. Recall that a subsystem version corresponds to one or more baselines.

In the single-step approach, the user selects a subsystem version, a baseline to which that subsystem version corresponds (the *source* baseline) and an immediate successor reference baseline (the *target* baseline). The SubCM Tool queries CM Synergy to see if any of the atomic components of the subsystem have changed since the source baseline. If so, it checks out new SCSs (with new version identifiers) for each of the subsystem components affected by the changes, updates them accordingly, and creates associated change descriptions (see Fig. 13); the new SCS corresponds to the target baseline. If the subsystem has not changed between the source and target baselines, the target baseline is simply added to the baselines window in the Configuration View.

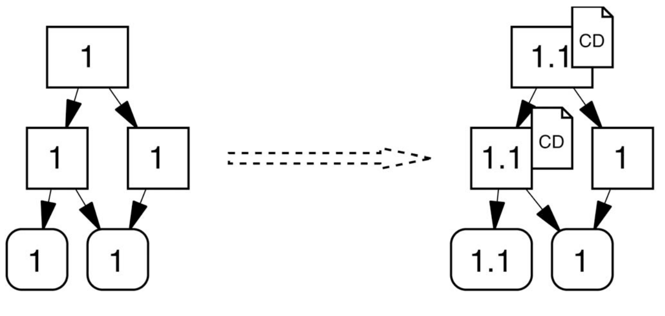


Fig. 13. Generation of new SCSs and Change Descriptions (CD) after a change to an atomic CI.

In the fully automated approach, the user selects a source baseline only and the tool automatically updates with respect to *all* baselines downstream from the selected source baseline. This process may take a couple of minutes, depending on the size of the hierarchy and the number of downstream baselines. The generated data (SCSs and change descriptions) is therefore stored locally by the SubCM Tool to facilitate quick navigation later in the Configuration and Change Views. Thus, the overhead incurred by the tool is incurred only once per baseline and hierarchy.

5.3 Integration into the Software Lifecycle

While all the features discussed above were used in the evaluation of the SubCM Tool at Invensys (see Section 4), not all of the features discussed in this section below were evaluated. This section presents some of the potential uses that we identified for the tool within the software development process, and as such a rationale for many of SubCM Tool's features. Section 6 discusses which features have actually been used in practice and Section 7 compares the features of the SubCM Tool to related research work and widely used existing CM tools.

5.3.1 Supporting the Build Process

The constituents of subsystems change in the course of development and maintenance activities. These changes affect the compiled executable of a product when a new build is performed, which is done at regular intervals. The build process is performed by deciding which atomic changes (represented by completed tasks from CM Synergy) will be incorporated into the new product version. This results in the reconfiguration of the product and its subsystems. The incorporated changes will already have been tested to some extent.

A completed task is included in the new configuration if the responsible team leader has reviewed and accepted the task and a test build has succeeded. The change should then be documented by updating the SCS and the change description.

The build manager can be assisted in the selection of tasks to include in a new release by browsing the variant comparison of the latest release of that product with respect to the most recent stable baseline (to find the most recent versions of each atomic CI).

5.3.2 Release Integrity Checks

An important purpose of CM is to ensure the integrity of a product release, i.e., that all artefacts in the configuration are consistent with the changes intended for the release. Integrity checks may be applied for individual change requests to ensure that changes have been reviewed and tested and that all associated tasks are incorporated into the product release. Conversely, the product release should incorporate only tasks that are associated with the relevant change requests. Of course, it is not sufficient to simply check task status: Manual effort is still needed to check the technical completeness of each change and to ensure that functional specifications, user manuals, and test specifications have been updated in accordance with the details recorded in the change request. However, the SubCM Tool simplifies the check by enabling affected artefacts to be browsed and to compare the task identifiers associated with a given change request to be checked against those actually incorporated into the release.

Our approach can also assist with integrity checks between subsystems. A common example is the check for consistency after modification of a subsystem interface. Such a modification affects several dependent subsystems and may therefore be implemented by different teams. By using the tool's facility to impose different subsystem hierarchies on top of the same physical structure, the tool may be used to coordinate the modification by sharing interface items between the dependent subsystems. Any change to these shared items would be flagged as a change in each of the dependent subsystems, thereby identifying areas for review and test activities to verify that the changes have been integrated into all the systems that share the interface. Another intersubsystem integrity check arises where code is replicated between subsystems, for example, where two variants of a function are used in different products. In this instance, a change to one of the items should be applied consistently to all related items. This activity is supported by enabling creation of a subsystem consisting of all the related items so that, if one of the items changes, the reviewer can check that all of the related items have been changed accordingly.

Additional confidence that the content and characterization of a release is correct can be provided by the tool. It can assist production of the release note that describes the public artefacts of the revised product and changes that have occurred since the previous release.

5.3.3 Product Support and Product Management

The SubCM Tool approach also assists with product support and product management. For example, it can assist planning of an upgrade path for customers wishing to correct reported system faults if those faults have already been corrected in ongoing development. In such situations, the tool allows the changes between existing and upgrade releases to be easily characterized to ensure that functionality is not lost in the upgrade or, conversely, that no undesired functionality is included.

The comparison of two arbitrary SCSs by showing the difference of their configurations is also supported. This facility can be used to determine whether all existing products still need to be supported or whether some products can be merged into a single product to reduce the complexity of the overall software configuration.

The data maintained by the SubCM Tool can also be used to collect measures of changes for each product release, including frequency, scope, and effort of document and source code changes. This data can be used to:

- predict the effort required on future releases and tasks,
- determine the extent of changes, that is: do change requests and tasks affect a small set of co-located objects or a large, distributed set of objects,
- identify a reasonable focus of verification efforts, namely, where subsystems have changed frequently (frequent changes to a part of the code may also indicate that a reengineering of that part would be useful), and
- profile source code stability to determine priorities for regression testing.

6 Experience

This section reports experience using the SubCM Tool with an Invensys SCADA product development group. We give examples of some of the subsystems to which the tool has been applied, and how the tool has been used to analyze patterns of subsystem change. The particular product examined is software for a Remote Terminal Unit (RTU), which exchanges digital and analog data with plant equipment and communicates processed data to a central control centre through a variety of protocols and media.

6.1 Extracting Change Data

Build managers, who are members of the SCADA V&V team, used the SubCM Tool to define subsystem hierarchies for two SCADA software products and to generate configuration and change data for the hierarchies using the tool. The products analyzed were two variants of embedded software performing analog and digital IO, data processing, and remote communications via multiple protocols. The CIs included some 300 KLOC of source code across 385 atomic CIs. Nonsource code CIs were not included. The subsystem definition resulted in 38 subsystems overall with 29 top-level subsystems defined. The maximum depth of the subsystem hierarchy was six levels. The hierarchy was refined through the definition and analysis process to provide greater insight into large or complex subsystems.

Configuration and change data was extracted on a number of product baselines. Fig. 14 shows a section of the version tree of product baselines used in the investigation. The Invensys SCADA group follows a baseline model that is similar to the branching-by-purpose model described by

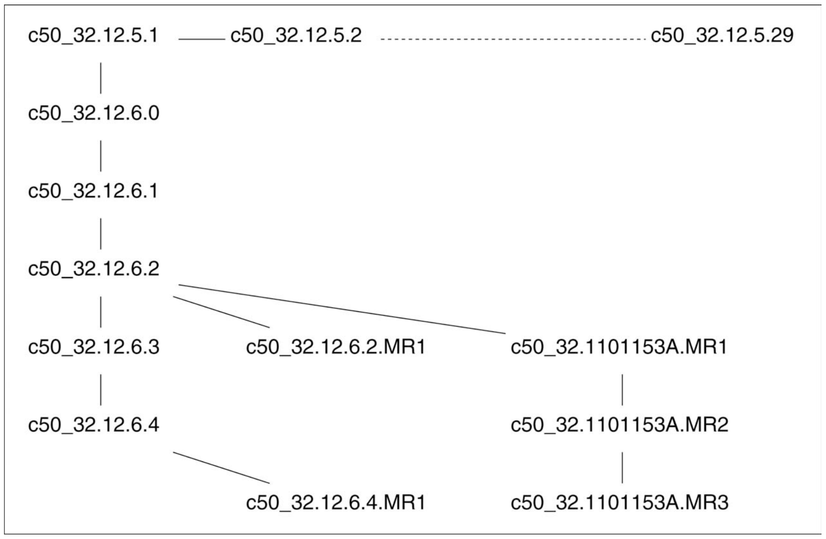


Fig. 14. Part of the baselines version tree in the CM repository.

Walrad and Strom [34]. Ongoing system development takes place in the main development path, with tasks integrated and tested on a regular basis. This is represented by the baselines c50_32.12.5.1 to c50_32.12.5.29. When all tasks that are scheduled for a product release are complete, a new product release is prepared in the product release branch. This is done by integrating the set of scheduled tasks from the main development path to a selected baseline in the product release branch. A main product branch is maintained for major product releases (as identified by c50_32.12.6.0 - c50_32.12.6.4). On occasion, subbranches are created in the product release branch to accommodate minor releases or variants developed for particular customers.

Configuration change data was extracted from the main development path using the SubCM Tool's automatic extraction utility applied to baselines c50_32.12.5.7 to c50_32.12.5.29 (for simplicity, these have been named versions 1 to 23 in Table 1). The resultant data in Table 1 shows the number of tasks applied to each subsystem in each development baseline.

The data presented in this section was obtained by small routines inside the SubCM Tool which process the data stored by the SubCM Tool, i.e., computing the data incurred negligible overhead. The graphs in the figures in the next sections were drawn from this automatically generated data using a standard LaTeX package.

TABLE 1
Number of Changes to Top-Level Subsystems between Versions 1 and 23 of Product 1101155, in Terms of Number of Tasks Incorporated

No	Subsystem name	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	Calculations							3	4	2					4								1
2	Foxcom												1										
3	ADI_Card								1														
4	Analog_Input_Card		1																				
5	Analog_Output_Card																						
6	C300_Slave								4	5													
7	Logger																						
8	Include_Files						1				1											1	1
9	IEC103_Master													2						2			
10	Wisp_Slave								4	1													
11	Wisp_Master								1														
12	TCP_IP							3								6		2		1		2	
13	Picore	1	1	1	1	1	3	5	13	5	2	1	3	2	3	6	1	1	3	1	1	2	7
14	OS_Files_X86		1						1			1	10	4	4	4		1	1				2
15	Optonet							2	4	1												1	3
16	Multi_IO								1														3
17	Modbus_Slave								3	1					1								
18	Modbus_Master											2		1	1								
19	Condensed_Package								1														1
20	Conitel_Master							3	6														
21	IEC101_Slave			2	3				9		1	4								2		20	
22	Conitel_Slave					3		3	6	6													
23	Digital_Output_Card																			1			
24	IDF								9														
25	DNP3	5	4	5	3		4		7	4	9		7		12	13	2		4			5	3
26	Flash_File_System												3										
27	Harris					2		4	7	4													
28	ACT_card					1			2														1
29	1101155.mif	1	1	1	1	1	2	1	4	1	1	1	1	1	1	1	1	2	4	1	1	1	1

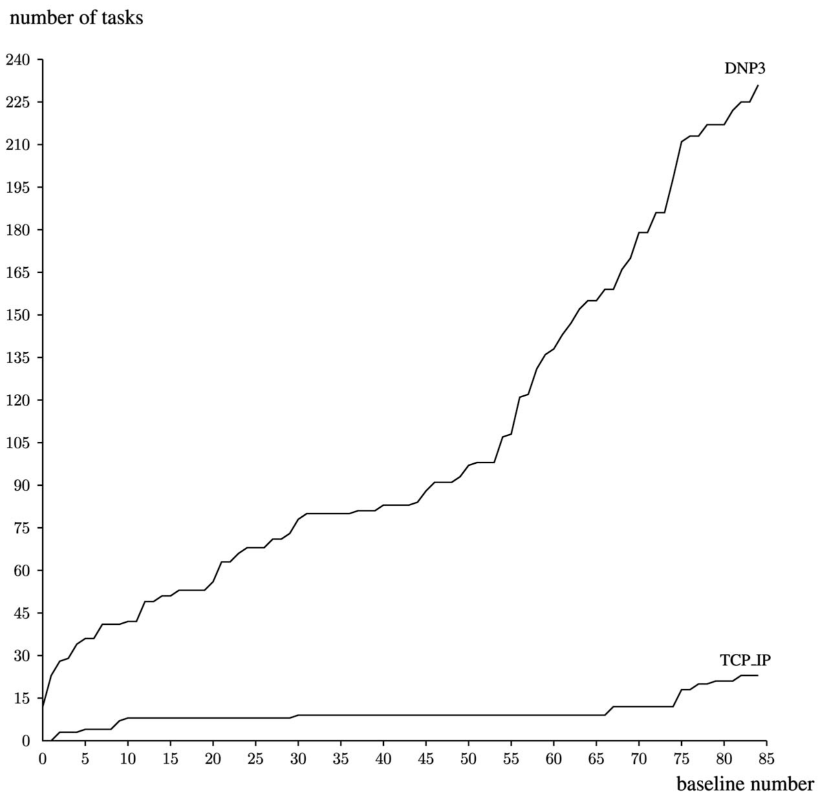


Fig. 15. Cumulative number of changes (in tasks) to TCP_IP (lower graph) and DNP3 (upper graph) over the 86 baselines.

6.2 Analysis of Change Data

The change data generated by the SubCM Tool was used to investigate patterns of change within subsystems and across development baselines. This process resulted in two refinements of the subsystem hierarchies. First, a file containing baseline version information was moved into a separate top-level subsystem. This was done to distinguish nonfunctional changes from the remaining software. Second, the largest subsystem, DNP3, was decomposed to further expose the source of changes. While this assisted the analysis somewhat, a significant number of changes were traced to individual, quite large, CIs. These were identified as candidates for refactoring to assist future impact analysis.

Analysis of change patterns identified baselines with widespread change associated with changes of common data structures. It also identified peaks of change in particular subsystems, subsystems with continual change as well as subsystems with very few or no recent changes. This information was a useful validation of staff feelings about subsystem change frequency and the subsequent focus of effort that had been in place in impact analysis and regression testing activities.

Two subsystems were selected for further investigation; DNP3, with a relatively high continuous pattern of change, and TCP_IP, with previous stability but more frequent change in recent history. In this instance, the baselines used in the analysis were extended back to the full history of the CM system. The cumulative changes for the subsystems are shown in Fig. 15. The amount of change on the DNP3 subsystem, represented by the upper graph in Fig. 15, identified a high-maintenance area of the system. This was accounted for by a continuous enhancement of the subsystem functionality to accommodate evolving standards and usage profiles. Potential actions identified for the subsystem were to undertake a requirements analysis of further likely enhancements and undertake them proactively, perform refactoring to lower the maintenance cost and/or invest further in automated regression testing to reduce reverification costs. Analysis of changes for the TCP_IP subsystem, represented by the lower graph in Fig. 15, identified that recent changes were made to accommodate an upgrade of hardware platform. Further monitoring will be performed to confirm that the subsystem settles into a period of stability again.

6.3 Product Releases

The SubCM Tool's variant comparison functionality was used to investigate how product releases related back to the main development path. We focussed on the DNP3 subsystem in the baselines c50_32.12.6.1 to 6.4 of the repository, which correspond to versions 3.1 to 3.4 of DNP3. To do so, we chose the most recent baseline common to the main development path and the product and used the tool's automated facility to populate the subsystem hierarchy with configuration and change data for all subsequent releases—both product releases and development releases.

The data in Fig. 16 was generated by comparing each product release with the baselines on the main development path. The four product releases occurred after development baselines 5, 12, 12, and 19, respectively—shown as vertical

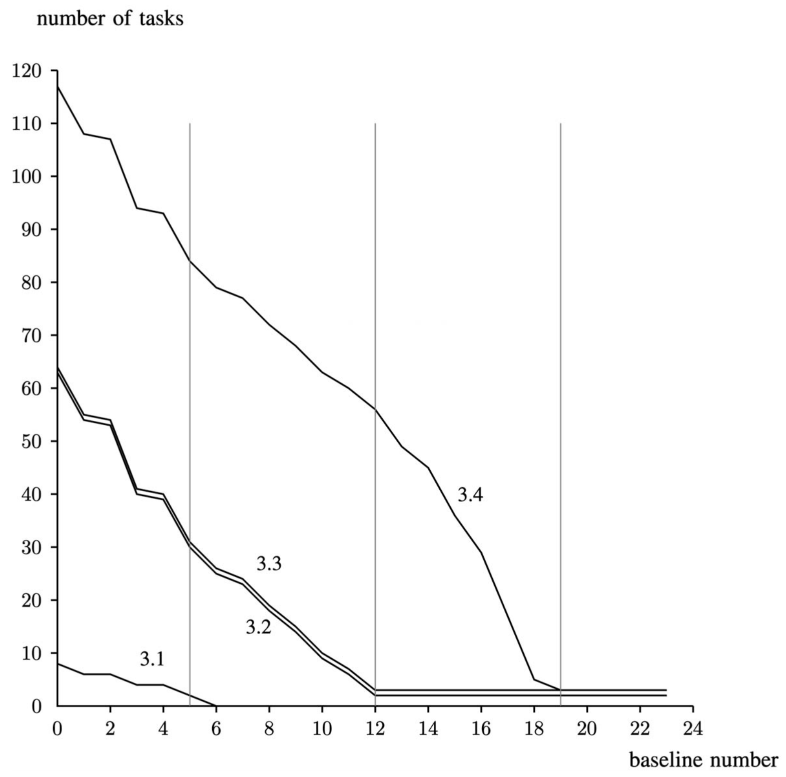


Fig. 16. Size of difference between product releases and development baselines (x-axis), in number of tasks yet to be incorporated (with respect to the given subsystem DNP3).

lines on the graphs. (Product release 3.2 was a release candidate that was rejected and followed immediately by a corrected version that became product release 3.3.) The graphs indicate that some tasks associated with product releases were incorporated into the main development path later or not at all. Analysis of the seeming anomalies revealed that tasks were incorporated into the release baseline to correct faults detected during the release testing. The changes were later integrated into the development baseline, either directly or indirectly by incorporating equivalent object changes into other tasks. While there were no issues with the product release integrity, the SubCM Tool provided a useful check and allowed the analysis to be conducted quite easily.

The data in Fig. 17 was generated by comparing the number of tasks in the development baseline against the number of released tasks. This comparison shows whether tasks included in the tested development baseline were not included in the release. The example shows that a number of DNP3 tasks were integrated into the development baseline to facilitate testing, but none of these were included in the 3.1 release (which occurred after baseline 5). Instead, they were combined with additional tasks and integrated as a whole into the 3.2/3.3 release (baseline 12). Since there was no partial change to the subsystem for the 3.1 release, no risk to release integrity was posed. The analysis also identified one task that was not included in the 3.4 release (baseline 19). Investigation revealed that this task was related to a change that was not ready for release so was correctly omitted.

6.4 SubCM Tool Current Use and Enhancements

Although the SubCM tool has influenced policy and practice at Invensys, the tool is not yet in widespread use there, for two main reasons. First, software design enhancements carried out subsequent to the tool evaluation have resulted in the consolidation of product variants into a single product; product baselines are also managed more carefully to reduce the effect of any branching. This has simplified the comparison of product versions for the purpose of product management and support, and reduced some of the original motivation for the SubCM Tool.

Second, change management administration has been improved by the introduction of Telelogic's Change Synergy. The tool allows product releases to be characterized by a set of change requests, with each change request managed by a defined lifecycle and approval process. Change requests are linked via the underlying CM system to the affected CIs and can be assigned a variety of attributes, including associated subsystems and releases.

These process improvements have partly addressed some of the advantages that would have been gained by the SubCM Tool. For example, change data and patterns of change can be studied by categorizing change requests into subsystems and releases. This would allow the change summary data in Table 1 to be extracted for the subsystems nominated. It would also be possible to trace changes to particular configuration items, although this would be more cumbersome. Integrity checks are built into the lifecycle of each change request to ensure that modifications are complete, consistent, and correct. Configuration audits performed for each release ensure that all changes

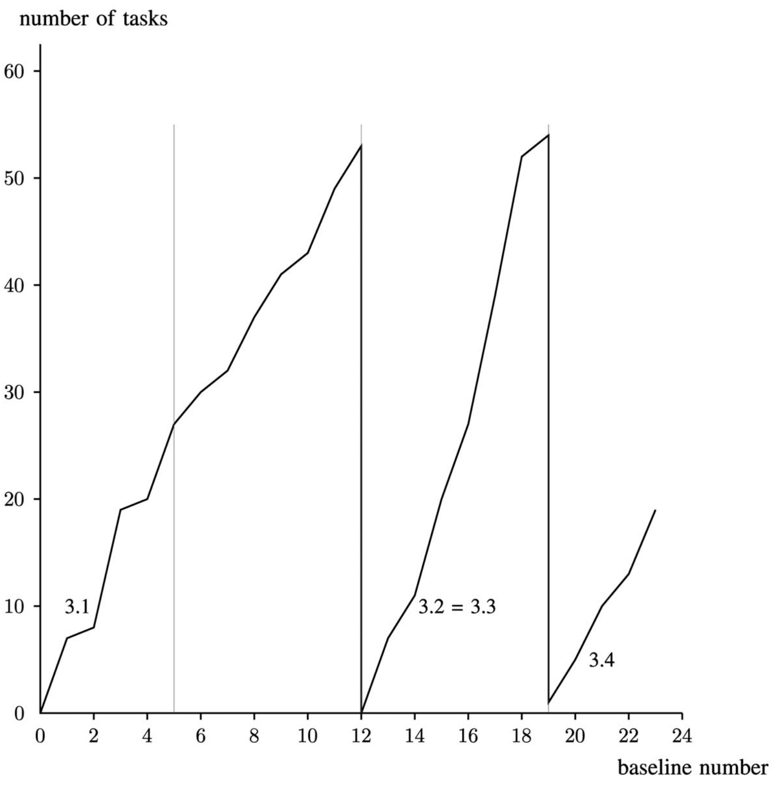


Fig. 17. Number of tasks in development baseline not yet put into released products (with respect to the given subsystem DNP3).

scheduled for the release have been incorporated into the release baseline.

Nevertheless, some advantages of the SubCM tool remain unrealized. Primarily, these are related to the configuration view described in Section 4.2, which cannot easily be replicated by a pure change management system. This view would be useful for visualizing changes from the perspective of the product CI structure, particularly since it offers a richer and more automated representation of subsystem structure than the manual association of change requests to subsystem identifiers. For example, the SubCM Tool would make it easier to define subsystem hierarchies in order to analyze change at multiple levels of detail, or define multiple system hierarchies to represent dependencies between CIs. Since the subsystem hierarchies are built from the actual CIs, they would also provide an accurate representation of change that can be used for more reliable impact analysis. While this potential was never fully explored in an industrial context, it is considered that this would be a valuable extension to the existing industrial tool set.

The experience of using the tool also identified some potential enhancements to aid analysis of change data:

1. Record metrics of the size of changes, for example, by the lines of code added, deleted, or modified. This would improve analysis of the effort involved in software maintenance.
2. Record the cause of change, for example, to distinguish between enhancements, corrections in response to external fault reports, and rework of unreleased changes in response to code inspections or internal testing.

In both cases, the ability to focus on retrospectively defined subsystems is important, so analysts can investigate what was actually changed rather than simply what was planned to be changed.

7 Related Work

This section describes related research on hierarchical CM and compares the SubCM Tool's capabilities with those of a range of widely used CM tools.

7.1 Research

Estublier and Casallas [16] discussed the need for structuring beyond the structuring provided by file systems and implemented their ideas in the Adele Configuration Manager. They argued that a software CM system must provide both an object management system with a powerful modeling capability (and its associated complexity), as well as basic files and directories with limited modeling power. The authors conclude that a software CM system should have three components: an object manager controlling the repository, a workspace manager controlling the work areas of developers, and a process manager controlling task and activities across the other two components. Adele was used within industry in the early 1990s as a foundation for in-house software engineering environments and process control. The SubCM Tool is motivated by a similar desire, to see a more powerful mechanism for modeling CM repositories than that provided by file systems and existing tools. However, their approach is different in that it is based on objects and provides a different way of approaching CM. Rather than replacing current CM practices and tools, our approach has been to provide add-on support to existing tools.

Asklund et al. [1] propose a unified extensional versioning model. Traditional CM versioning is done extensionally at the atomic level and intentionally at the composite level. In their proposed framework, extensional versioning is used at both the atomic and the composite level. In the model, a *document* is a tree structure consisting of local data, a composite node (which is a collection of nodes), or a link node (which is used to represent arbitrary relations between documents). This model supports our notion of subsystems as composite nodes and is more general because it allows arbitrary relationships between documents to be established. It also allows fine-grained CM, in that atomic nodes (which contain just data) can be parts of documents rather than documents themselves. The model supports and manages change by creating and closing revisions of documents, i.e., when an atomic node is updated, this creates a new version of all the documents that are linked to it. Changes between documents are not explicitly recorded, but can always be calculated because all past versions of documents are available.

The model has been implemented in three tools: COOP/Orm supports development in distributed groups, CoEd is a research prototype that supports the writing and editing of hierarchically structured documents, and Ragnarok is a software development environment that uses software architecture as a framework for version and CM. While Ragnarok has seen industrial usage, COOP/Orm and CoEd have only been used in a research environment. COOP/Orm, CoEd, and Ragnarok are discussed in more detail below. Asklund et al. note that the versioning model does not support product variants. It is also not clear how their model can be used retroactively to analyse existing document and code repositories, and the comparison of variants is only supported through a recursive "diff" on atomic nodes as opposed to the more structured comparison supported by the SubCM Tool.

COOP/Orm [20] supports development in distributed groups and, as such, has support for fine-grained version control and advanced merge facilities. The motivation for this work is quite different from our work. It is mainly aimed at developers working in a collaborative environment. The creation of subsystems through the binding of documents would be clumsy for the flexible creation of subsystems that we are interested in. Moreover, the integration of the CM facilities in the document editing environment would have a significant impact on the way in which developers work, which is something that we tried to avoid.

CoEd [3] provides support during the cooperative development of hierarchical documents. The approach focuses on document units (chapters, sections, etc.) and the tracking of changes to these during editing. A group of components can be versioned (called metaversioning) and documents can be created by selecting different versions of these units and/or metaversions. The tool is intended to support the selection of variants of each unit/metaversion that comprise a specific version of the overall document, but does not support the reorganization of documents.

Ragnarok [6] is a software development environment that uses software architecture as a framework for version and configuration management. Ragnarok has been applied on three substantial projects, including Ragnarok itself. Data has been collected on these projects in the form of open-ended interviews and analysis of the RCM (the CM module in Ragnarok) logs. The results reported are that the model feels natural to developers, the versioned documents become the focus, the model and tool provide a traceable architectural evolution, and the intermediate revisions of software components that are created by the tool are not perceived as a problem. The emphasis in this model and the tool is on software architecture, and the tool is mostly intended for developers. There are no explicit change descriptions, but architectural differences are supported by allowing differences between versioned documents to be computed in a recursive manner.

Lin and Reiss [18] present a framework for programming environments that handles versions and configurations directly in terms of the functions and classes in the source code. It emphasises system building and version control issues and discusses how the framework supports these, focusing on issues in software reuse and cooperative programming. The model is built around software units, which are typically software functions or classes, but can be higher level software artefacts as well. The notion of a subsystem is also used, but in this case that means all the software units that are linked from a given software unit, either directly or transitively. Change management of software units or subsystems is not discussed.

Their framework is implemented in the prototype environment POEM (Programmable Object-centered EnvironMent), which provides an interface that lets the user view software units, subsystems, and links between them. It also provides a number of editors for the interface, implementation, and documentation of software units. The environment stores the source code and the links between various software units itself and, as such, cannot be used to manage an existing code base (it is aimed at supporting development of new programs). It also does not support documents that do not contain source code, such as requirements or design documents. POEM does not seem to have been used in an industrial setting.

Conradi and Westfechtel [9] summarize existing *version models* for software configuration models and propose a framework that, while recognizing problems associated with hierarchical systems, focuses primarily on new methods of object-based versioning. This work is based on the earlier work of Conradi [8] on EPOS (Expert System for Program and System Development), a software engineering environment (SEE) with emphasis on process modeling, CM, and support for cooperative work. In contrast, our approach is a lightweight add-on to existing CM techniques. Given the investment embodied in existing CM repositories and the effort required to port legacy CM data to new databases, we believe our lightweight approach is a cost-effective alternative to the complete redesign that would be involved in changing CM systems.

Recently, Conradi and Westfechtel [35] have taken a more component-based approach and are focusing on the overlap between software architecture and software CM. They state that, while software architecture and software CM play different roles within the software life cycle, they overlap in the product and version spaces and that software CM tools should take software architecture into consideration, but not be driven by it. As such this is orthogonal to our work on the SubCM Tool, which while also component-based, is focused on component understandability, not component-driven development.

NUCM (Network-Unified CM) [30], [31] is a testbed rather than a tool that provides basic CM operations in a distributed environment and allows the separation of CM repositories from the policies that control them. This separation is achieved by providing a generic model of a distributed repository and a programmatic interface to the repository. In their generic model, they propose a storage model using atoms and collections. These collections are equivalent to our subsystems as they provide a mechanism for structuring that is independent of underlying structure and can be arbitrarily nested. The work diverges at this point as van der Hoek et al. are focused on providing a new approach to CM using a CM-specific distributed repository and we are focused on improving visibility and understandability of systems by using subsystems. It would be possible to implement our system within the testbed provided by van der Hoek et al. and remove the localized aspect (restricted within a local CM repository) of our work.

Our SubCM Tool can aid build managers in deciding which files, components, etc., are included in a particular version of the system under CM. The SubCM Tool allows this process to be separated from the physical structure of the software repository. The concept of disentangling the build process from the repository/software structure is being approached from non-CM directions as well. Source tree composition [14] achieves this disentanglement by providing a means of automating the assembly of software systems from reusable source code components and involves the integration of source code hierarchies, build processes, and configuration processes.

Our approach provides a subsystem-based way of visualizing the changes to a repository. Others provide different ways of viewing these changes. Eick et al. [15] describe a range of different ways of visualizing software changes, and how visualization can support investigation of the software change process. Our subsystem-based model could be extended to extract the kind of data used in their approach (such as who initiates change requests, who implements particular changes, and how much effort is involved) and extended to employ their visualization techniques. Lanza's "evolution matrix" [17] is a way of visualizing how OO software evolves, by using a matrix in which columns represent system versions, rows represent classes, and cells display a user-specified class metric, such as number of methods or number of instance variables. The paper introduces terminology from astronomy to describe the different ways in which classes evolve, such as pulsar, for a class that grows and shrinks repeatedly during its lifetime. Our work has focused on structuring and subsystems in particular to help understand change in a large software-based system. Barkstrom [2] shows how graphs can be used to understand and investigate large systems, in this case, scientific satellite data.

7.2 Tools

A selection of existing widely used tools [7] were investigated and evaluated for their ability to support the SubCM approach. The focus was on their ability to support arbitrary levels of subsystems, whether these subsystems could be defined independently of the physical repository structure, and how easily the subsystems could be changed. Furthermore, the versioning of these subsystems was investigated with specific focus on subsystem histories, the update and aggregation of these histories, and the comparison of subsystem versions. While many of the tools offer some subsystem tracking facilities if the desired subsystem can be defined in advance, they currently have little or no capability to extract information when subsystems are defined retrospectively. Moreover, versioning and variant comparison of subsystems is only supported to a limited extent (typically a recursive diff).

Though not a commercial tool, CVS [12] is heavily used in industry. CVS has modules that provide some of our subsystem features. Modules, which can be arbitrarily nested, define the contents of a view over the repository. Module definitions typically contain directories, but may contain other modules and files. Modules give some ability to have structure different to the stored structure, but to be viewed, modules of interest must be extracted from the repository and viewed using external tools. Modules themselves are not versioned, though it would be possible to *tag* (label) all the constituents of a specific release of a module to achieve a degree of versioning. Comparison of module versions could likewise be achieved by checking out the versions of interest (via tags) and then using tools such as diff.

Continuus Change Management [11], [10] uses projects and directories to structure its repository. More recently, CM Synergy [28] (now a Telelogic company) has added subprojects. The difference between projects and directories is subtle and the main point is that projects can contain subprojects and directories, but directories cannot contain projects or subprojects and subprojects cannot contain projects. Further, projects are intended to model complete systems, subprojects to model logical components, and directories are used for further structuring. The projects/subproject combination give arbitrary depth of subsystems and can be changed with ease. All projects and directories are versioned, but no other information is stored or visible at the subsystem level and only directories can be compared.

Razor [32] predominately uses folders, which are directories and not versioned, for structuring. Razor does support the concept of *threads* which are a build management tool. Files and/or folders are allocated to a thread and as such give different views over the repository. These views are not to arbitrary depths as standard threads cannot contain other threads, but there does exist a special thread called "Project" that can contain a collection of standard threads, but not other configuration items. Within this two-level structure, threads are easily changeable. Threads are versioned and a brief message describing the version can be recorded, but no summary of how constituents (simple items, folders, or threads) change is available. Thread versions can be compared, but only by recursively showing all the differences of every file in every subsystem (folder and/or thread), similar to a recursive diff.

Rational's ClearCase [24], [25], [26] is similar to Razor in its handling of subsystem structuring. ClearCase has a restricted subsystem-based approach. The top level consists of a project, which can contain components, and components can contain ordinary directory structures. Projects and components are versioned by explicitly associating a baseline with a component, where a baseline is a collection of files (and the version of interest). Projects are flexible and can be added easily with shared constituents, but have limited depth as with Razor. ClearCase is more restrictive as projects can only contain components, and components must map to directories and everything within that directory belongs to the component. Components are versioned, but no other history is explicitly stored about the component. Project and component comparisons are similar to those available in Razor.

Microsoft's Visual SourceSafe [21] provides a subsystemming mechanism that goes beyond that of the other tools. Visual SourceSafe uses projects which can be nested to a depth of 15 (no reason for this restriction is given); however, there is a special root project. All directories are automatically projects, but projects can be added that do not map directly to directories and objects can be shared between projects, supporting differing views on the same repository. Projects are versioned, but a project's version increments every time a constituent is added, deleted, changed, etc. Label-based (or tags using CVS terminology) versioning of projects is also supported. The version history for a project simply records the sequence, but not details of how constituents changed. Projects can be compared using either version numbers or labels, but due to the fine-granularity of version numbers, label-based comparisons are more effective. All comparisons are flat comparisons as already described for Razor and ClearCase.

The tools evaluated do provide some of the functionality of the SubCM Tool, especially some form of intermediate structuring. However the versioning of these subsystems with specific focus on subsystem histories, the update and aggregation of these histories and the comparison of subsystem versions is limited or nonexistent. It would be possible to integrate our tool with all of the tools evaluated, except for Microsoft's Visual SourceSafe, and provide this more powerful and sophisticated approach to managing subsystems. The integration would be achieved using the command-line interface provided by these tools in much the same way that the SubCM Tool currently integrates with CM Synergy. Microsoft's Visual SourceSafe does not provide this type of command-line interface.

8 Conclusions

In summary, this paper introduced a generic lightweight tool to enable flexible configuration management (CM) of hierarchies of subsystems, designed to be used on top of more traditional file-based CM systems. The focus was on configuration identification and change tracking for user-defined subsystems, by automated extraction of configuration and change data from an underlying CM repository. The SubCM approach is lightweight because it requires no changes to be made to the system or repository structure, nor to existing CM practices. Furthermore, this lightweight approach is flexible as the cost of changing subsystem structure is minimized. A prototype instantiation of the tool was built to interface with Telelogic's CM Synergy tool and evaluated within a software development unit of Invensys. The SubCM Tool was compared with a range of existing CM tools and found to offer a more powerful and sophisticated approach to managing subsystems. While most existing tools provide some support for tracking development histories of subsystems, they all require that the subsystem be defined in advance; by contrast, the SubCM approach provides the "benefit of hindsight" by enabling subsystems to be defined retrospectively and their histories to be generated for selected points in their system's history. This approach could be usefully combined with advanced ways of visualising software changes [15].

The paper argues that the lightweight nature of the tool and the visibility it gives to subsystem configuration and change data enables a range of new process improvement capabilities.

The approach enables a range of product quality metrics to be tracked on a subsystem-by-subsystem basis. For example, the change history of a user-defined subsystem can be extracted automatically to reveal how often the subsystem changed and by how much. In the example, change data could be extracted at product level (in terms of numbers of atomic changes), at project level (number of tasks), or at "customer" level (number of change requests). Such data can reveal problem areas and hot spots, and suggest areas where redesign could usefully take place.

The approach also offers significant potential benefits for V&V teams. Although not yet currently implemented in SubCM Tool, the approach enables CM data to be integrated across multiple CM repositories. For example, documentation can be associated with the source code files to which it relates, and their status can be tracked jointly as a subsystem: If one changes without the other, this will become immediately obvious from the change history for that subsystem.

Another potential benefit is more efficient integrity checking for product releases. A subsystem can be established for a particular product, and a product release candidate can then be compared against an earlier product release (e.g., the last version that was released to a particular customer) to check whether the implemented changes actually agree with the planned changes, change request by change request.

Finally, the approach enables review and testing to be more focused, for example, by revealing which subsystems have changed and supporting the analysis of the impact of change requests.

Acknowledgments

The authors gratefully acknowledge the constructive comments of Paul Ellis, Alena Griffiths, Mark Staples, and Jason McDonald. This work was supported by Australian Research Council grant C49937058 (automated support for verification and validation of control system software).

References

[1] U. Asklund, L. Bendix, H.B. Christensen, and B. Magnusson, "The Unified Versioning Model," *Proc. Ninth Int'l Symp. System Configuration Management,* 1999.

[2] B. Barkstrom, "Software Architecture and Software Configuration Management," *Proc. Conf. Software Configuration Management (SCM 2001/2003),* B. Westfechtel and A. van der Hoek, eds., 2003.

[3] L. Bendix, P.N. Larsen, A.I. Nielsen, and J.L.S. Petersen, "Coed—A Tool for Versioning of Hierarchical Documents," *Proc. Eighth Int'l Symp. System Configuration Management,* 1998.

[4] P.E. Bennett, "Small Modules as Configuration Items in Certified Safety Critical Systems," *Proc. Sixth Safety Critical Systems Symp.,* F. Redmill and T. Anderson, eds., 1998.

[5] E.H. Bersoff, V.D. Henderson, and S.G. Siegel, *Software Configuration Management.* Prentice Hall, 1980.

[6] H.B. Christensen, "The Ragnarok Architectural Software Configuration Management Model," *Proc. 32nd Hawaii Int'l Conf. System Sciences,* 1999.

[7] "CM Yellow Pages," http://www.cmtoday.com/, 2003.

[8] R. Conradi, M. Hagaseth, J. Larsen, M. Nguyen, B. Munch, P. Westby, W. Zhu, M. Jaccheri, and C. Liu, "Object-Oriented and Cooperative Process Modelling in EPOS," *Software Process Modelling and Technology* A. Finkelstein, J. Kramer, and B. Nuseibeh, eds., Advanced Software Development Series, pp. 9-32, 1994.

[9] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys,* vol. 30, no. 2, pp. 232-282, June 1998.

[10] "Task-Based CM,"Continuus Software Corporation, 1996.

[11] "Introduction to Continuus/CM,"Continuus Software Corporation, 1999.

[12] CVS, "Concurrent Version System," http://www.cvshome.org, 2003.

[13] S. Dart, "Concepts in Configuration Management Systems," *Proc. Third Int'l Software Configuration Management Workshop,* pp. 1-18, June 1991.

[14] M. de Jonge, "Source Tree Composition," *Proc. Seventh Int'l Conf. Software Reuse,* C. Gaeck, ed., 2002.

[15] S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster, "Visualizing Software Changes," *IEEE Trans. Software Eng.,* vol. 28, no. 4, pp. 396-412, Apr. 2002.

[16] J. Estublier and R. Casallas, "The Adele Configuration Manager," *Configuration Management,* W. Tichy, ed., 1994.

[17] M. Lanza, "The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques," *Proc. Fourth Int'l Workshop Principles of Software Evolution,* pp. 37-42, 2002.

[18] Y.-J. Lin and S.P. Reiss, "Configuration Management with Logical Structures," *Proc. Int'l Conf. Software Eng. 18,* pp. 298-307, Mar. 1996.

[19] P. Lindsay, A. MacDonald, M. Staples, and P. Strooper, "A Framework for Subsystem-Based Configuration Management," *Proc. Australian Software Eng. Conf.,* D. Grant and L. Sterling, eds., pp. 275-284, 2001.

[20] B. Magnusson and U. Asklund, "Fine Grained Version Control of Configurations in COOP/Orm," *Proc. Sixth Int'l Symp. System Configuration Management,* pp. 31-48, 1996.

[21] Microsoft, "Visual SourceSafe," http://msdn.microsoft.com/ssafe, 2004.

[22] A. Midha, "Software Configuration Management for the 21st Century," Technical Report 2(1), Bell Labs Technical J., Winter, 1997, http://citeseer.nj.nec.com/midha97software.html.

[23] Python Software Foundation Offical Website, http://www.python.org, 2003.

[24] Rational, "ClearCase,"http://www.rational.com, 2004.

[25] Rational, "Introduction to ClearCase,"Number 800-012697-000 in Rational Technical Series, Rational Software Corporation, Dec. 1999.

[26] Rational, "Unified Change Management from Rational Software: An Activity-Based Process for Managing Change,"White Paper TP-710 10/00, Rational, the e-development company, Oct. 2000.

[27] J.A. Scott and D. Nisse, "Software Configuration Management," *Guide to the Software Eng. Body of Knowledge—Trial Version,* A. Abran and J.W. Moore, eds., chapter 7, pp. 103-120, 2001, http://www.swebok.org/.

[28] Telelogic, "CM Synergy Toolset," http://www.telelogic.com/products/synergy/cmsynergy, 2003.

[29] U.K. Ministry of Defence, "Configuration Management of Defence Material," Defence Standard 05-57/Issue 4, July 2000, http://www.dstan.mod.uk.

[30] A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. Wolf, "A Testbed for Configuration Management Policy Programming," *IEEE Trans. Software Eng.,* vol. 28, no. 1, pp. 79-99, Jan. 2002.

[31] A. van der Hoek, D. Heimbigner, and A. Wolf, "A Generic, Peer-to-Peer Repository for Distributed Configuration Management," *Proc. Int'l Conf. Software Eng. 18,* pp. 308-317, Mar. 1996.

[32] *Razor: Release management, File Version Control and Problem Tracking,* Visible Systems Corporation, 2003, http://www.visible.com/Products/Razor/index.html.

[33] H. Völzer, B. Atchison, P. Lindsay, A. MacDonald, and P. Strooper, "A Tool for Subsystem Configuration Management," *Proc. Int'l Conf. Software Maintenance,* pp. 492-500, 2002.

[34] C. Walrad and D. Strom, "The Importance of Branching Models in SCM," *Computer,* pp. 31-38, Sept. 2002.

[35] B. Westfechtel and R. Conradi, "Software Architecture and Software Configuration Management," *Proc. Conf. Software Configuration Management (SCM 2001/2003),* B. Westfechtel and A. van der Hoek, eds., 2003.

[36] D. Whitgift, *Methods and Tools for Software Configuration Management.* John Wiley and Sons, 1991.

Hagen Völzer received the master's (1995) and doctoral (2000) degrees in computer science from Humboldt-University Berlin, Germany. He has been a research fellow at the Software Verification Research Centre at the University of Queensland, Australia (2001-2003), and he is currently a senior research and teaching associate at the University of Lübeck, Germany. His research interests include systems engineering, formal methods, distributed computing, concurrency theory, and Petri nets.

Anthony MacDonald received the BSc (hons) and PhD degrees from The University of Queensland in 1993 and 1998. He is a lecturer in the School of Information Technology and Electrical Engineering at The University of Queensland. Prior to joining academia, he was a Research Officer at the Software Verification Research Centre at the University of Queensland, Australia (1998-2000) working on Software Development Environments. His research interests include software development environments, software architecture, software design, and, more recently, Model-Driven Architecture/Development.

Brenton Atchison received the PhD degree from the University of Queensland. He has over 11 years experience in critical software systems development and assurance. He is currently the assurance manager for Invensys Rail Systems, Australia. Prior to his current role, he worked as a consultant and engineer in the defense, transportation, and civil infrastructure domains. He is a member of the IEEE.

Andrew Hanlon has degrees in electrical engineering and computing from the Queensland University of Technology. He has worked in critical softare systems for three years with the first two years as a software verification engineer.

Peter Lindsay is Boeing Professor of Systems Engineering at the University of Queensland, Australia, and director of the ARC Centre for Complex Systems. He is coauthor of two books on formal specification and verification of software systems. In recent years, he has been involved with safety and security critical applications in areas such as air traffic control, embedded medical devices, ship-board defense, emergency service dispatch systems, and an international diplomatic network. His current research interests include techniques for the analysis, development, and assurance of complex, network-based systems.

Paul Strooper received the BMath and MMath degrees in computer science in 1986 and 1988 from the University of Waterloo and the PhD degree in computer science in 1990 from the University of Victoria. He is an associate professor in the School of Information Technology and Electronic Engineering at The University of Queensland. His research interests include software engineering, especially software specification, verification, and testing, and logic programming, especially program transformation and refinement. He has had substantial interaction with industry through collaborative research projects, training, and consultation in the area of software verification and validation. He was one of the program cochairs for the 2002 Asia-Pacific Software Engineering Conference and the program chair for the 2004 and 2005 Australian Software Engineering Conferences. He is a member of the IEEE.