SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 02-07

A Tool for Subsystem Configuration Management

Hagen Völzer **Brenton Atchison**
Peter Lindsay **Anthony MacDonald**

Paul Strooper

March 18, 2002

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available from `http://svrc.it.uq.edu.au`

A Tool for Subsystem Configuration Management

Hagen Völzer* Brenton Atchison† Peter Lindsay* Anthony MacDonald‡
Paul Strooper‡

Abstract

This paper describes a tool that manages a hierarchical, "is a subsystem of"-structure on a set of software development artefacts and that provides configuration management (CM) for subsystems by interacting with an existing CM tool. The tool is based on a recently proposed framework for subsystem-based configuration management. The tool demonstrates the feasibility of the framework and develops it further. The design of the framework and the tool was developed in collaboration with Invensys SCADA Development and it is discussed in relation to their current software development process.

1 Introduction

1.1 Overview

Configuration Management (CM) [2, 7, 19, 21] is a key discipline for development and maintenance of large software systems. CM is concerned with controlling and recording the evolution of all software development artefacts, not just source-code control. Existing CM tools and methodologies are limited in the support they provide for configuration and change management of hierarchically structured systems.

This paper describes a tool – the *SubCM Tool* – which supports subsystem-based CM. By subsystems we mean logically coherent collections of software development artefacts, including code, documentation and test sets. Subsystems can contain other subsystems, and artefacts can be shared between different subsystems. The approach is generic and designed to be used on top of standard CM tools for managing basic artefacts, while enabling CM for subsystems.

The approach characterises system configurations by showing which versions of which components and subsystems make up a configuration. It improves the visibility of system changes, by revealing how individual subsystems have changed. We claim that the tool will aid in coordination of changes across different developers and different teams, and that it will facilitate system-integrity checking and product-upgrade planning, amongst other things.

This paper describes the rationale behind the SubCM Tool and describes a prototype together with examples of its potential use.

1.2 Motivation and background

Our framework and tool were developed in collaboration with the Invensys SCADA Development group, who builds Supervisory Control and Data Acquisition systems. The software of these systems consists of a considerable number of communication protocols, calculation functions and control interfaces. Invensys SCADA Development currently employs a range of software configuration management tools and processes. The framework and tool described in this paper were designed to support these processes and improve their productivity and effectiveness. We believe the framework and tool would be useful in any organisation that has to deal with developments involving complex configurations. Typically this means developments with multiple product families, where each product has a configuration consisting of large numbers of artefacts under individual version control, together with cross-artefact relationships such as traceability matrices. Different product families share components, and the one artefact may be used in different versions of different products. Many artefacts exist in variants for different hardware and/or tailored for different customers. This situation becomes even more complex over time: as products evolve and more customers are supported, it is necessary to support versions of products using old versions of components. Because different customers have differ-

*Software Verification Research Centre, The University of Queensland, Brisbane, Qld 4072, Australia. email: {voelzer, pal}@svrc.uq.edu.au

†Invensys SCADA development, PO Box 4009, Eight Mile Plains, Qld 4113, Australia. email: brenton.atchison@invensys.com

‡School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Qld 4072, Australia. email: {anti, pstroop}@itee.uq.edu.au

ent requirements, many different configurations may need to be supported. It is assumed that a Configuration Control Board, or some equivalent, oversees change management and a software build process is in place which integrates source code changes from multiple developers. A mature source code and file repository is typically used, such as Telelogic's Continuus Configuration Management (CCM)[1] methodology and toolset [18].

In such environments, CM is most evident at two levels:

– at the level where items such as code files, documents and test sets are under CM. Such items will be called *atomic Configuration Items* (CIs) in what follows; and

– at the level of products (whole systems) released to customers.

Subsystem CM introduces the potential for intermediate levels of CM. Candidates for subsystem CM include: very coarse-grained items such as product families and their major components; medium-grained components such as the modules implementing particular communications protocols; and smaller, shared components such as a code library for particular calculation functions. Fig. 1 shows the relationship of subsystems with products and atomic CIs.

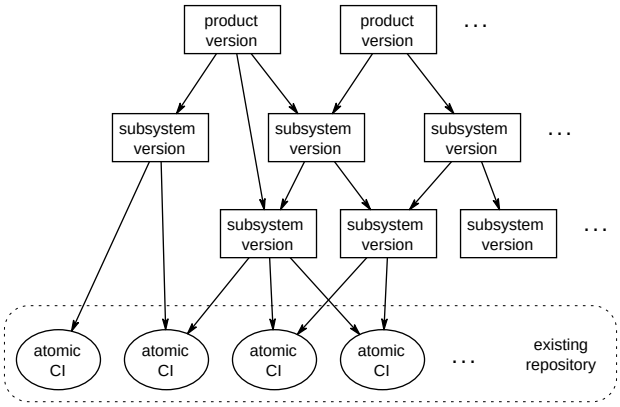

Fig. 1: Hierarchically structured configuration items

The following issues need to be taken into account in designing an approach to subsystem CM:

– The atomic CIs that make up a subsystem are possibly physically distributed across different CM repositories, and different CM practices may apply to different types of artefact (e.g. software development and system testing may be performed by different teams). Some CIs may not be held in a CM repository at all.

– Existing repositories are often rigid in structure and expensive to change. Their structure has typically developed over time and may be so ingrained in the company's practices that major restructuring is too costly to consider. As the examples above show, however, subsystems overlap and different kinds of subsystems are quite different in the way they are composed and viewed. The structure of subsystems should thus not be restricted to those imposed by the repository.

– A major investment, in time and money, has probably already been made in implementing a CM system for atomic CIs. Existing CM tools [6, 14, 15, 18] often allow aggregation of artefacts into "projects", which can be treated as subsystems. However, these tools do not support full versioning of aggregates, nor tracking of their change histories. Moreover, aggregation tends to be supported only along the lines of the file structure of the CM repository.

1.3 Approach

To overcome these problems we developed a general framework for subsystem-based CM that extends the capabilities of existing CM approaches [11]. The framework addresses the following issues:

– Characterisation: What is, and what is contained in, a given version of a subsystem? Conversely, which subsystem versions (if any) contain a given object.

– Change Descriptions: In what ways has a subsystem (and its components) changed between versions, and how are the differences recorded?

– Change Tracking: What caused a change to a subsystem, and how was the change carried out and checked?

The framework is described in more detail in Section 2 below.

The SubCM Tool has been developed to support configuration and change management within the framework, with minimal change to CM practices for atomic CIs. This paper describes how the tool will be integrated into software development, maintenance, and verification and validation (V&V) processes.

1.4 Related work

The problem of handling Configuration Items (CIs) at arbitrary levels of granularity has been recognised for some time [1, 21]. As noted above, however, existing CM

[1] Continuus Configuration Management is now called Telelogic CMSynergy.

tools provide little or no support for subsystem CM. Previous Software Verification Research Centre research explored management of fine-grained development artefacts and links between them [12, 16], but stopped short of considering hierarchical structures. The emergence of HTML and the world-wide web has increased the impetus for research into change management of highly interlinked artefacts [8], but full CM solutions are not yet available.

Lin and Reiss [10] describe an object-oriented approach to CM, whereby source-code functions and classes are put under version control. Their paper includes a discussion of version control issues for hierarchical systems, but does not provide a solution. Their prototype POEM system is built on top of an object-oriented database system, whereas our approach is independent of the underlying database technology. Christensen [3] describes an approach to configuration and version control of software artefacts with structural and dependency links. Conradi and Westfechtel [5] summarise existing version models for software configuration models that, while recognising problems associated with hierarchical systems, focus on new methods of object-based versioning. This work is based on the earlier work of Conradi [4] on EPOS (Expert System for Program and System Development), a software engineering environment (SEE) with emphasis on process modelling, software CM and support for cooperative work.

Configuration and change management of hierarchical structures is a less developed field. There is a growing body of research into adding version/revision control mechanisms to SEEs [4, 9, 13, 17, 20]. The main difference is that our framework is designed to be largely independent of the SEE and instead works on top of existing CM toolsets, with minimal change to underlying CM practices.

1.5 This paper

The paper is structured as follows. Section 2 describes a general framework for subsystem-based CM. Section 3 describes the SubCM Tool and Section 4 illustrates how the tool will be integrated into development practices.

2 The framework

This section presents the main ideas of the framework that was introduced in [11]. 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 framework supports the following three capabilities:

1. characterisation of the subsystem configuration, via *Subsystem Configuration Specifications (SCSs)*,
2. characterisation of the change between two consecutive versions of a subsystem, via *change descriptions*, and
3. tracking of what caused a change and how the change was carried out and checked.

These are explained in more detail below.

2.1 Subsystem configuration specifications

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. 2: SCS for a subsystem with three sub-subsystems

A *Subsystem Configuration Specification (SCS)* states which versions of which objects make up a particular version of a subsystem: see Fig. 2 for an example. An SCS consists of a subsystem identifier, which contains a name and a version number, 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", "source code", is associated with each constituent.

We assume that the being-a-constituent-of relation on subsystems forms a directed acyclic graph: i.e., a subsystem can be a constituent of several other subsystems, but we rule out circular dependencies (cf. Fig. 1). The DNP protocol subsystem in Fig. 2, for example, is a constituent of many products at Invensys SCADA Development.

2.2 Change descriptions

Fig. 3 shows a *change description* for the DNP protocol. A *change description* for two consecutive versions of a given

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).

Item	Change type	Description
Core	none	
Master	modified	Supporting documentation was updated.
Slave	modified	New functionality added and existing design modified to improve future maintainability.
Makefile.mak	none	

Fig. 3: Change description for version 3 of DNP

subsystem consists of the identifiers for both versions, 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 textual description of the change. The framework 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 configuration), *modified* (some content of the constituent has changed and the constituent appears in a different version in the new subsystem version), and *merged* (the constituent was derived by merging the former version with a parallel version). Fig. 3 shows that a change description provides an abstract description of how the subsystem has changed between two versions. More detail can be obtained by looking into the change descriptions of the constituents.

We also consider change descriptions for two non-consecutive 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. Currently, we aggregate change descriptions by concatenating their textual summaries as well as their lists of change items where redundant "none"-entries are deleted. This offers maximal information. Various concepts to suppress and reveal detail on demand can be implemented on top of this.

2.3 Change tracking

The third aspect of the framework concerns *change tracking*: i.e., recording of the reasons that a change was made and explanation of how and by whom it was carried out. Because organisations differ widely in how they do this, it is more difficult to offer generic support for this part of the framework than the preceding parts. The mechanism used here is very simple however: we associate with each change item a reference to how the change was made.

We assume two levels of change management: *change requests* and *tasks* (see Fig. 4). A *change request* is created when a bug 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 CCM. (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.

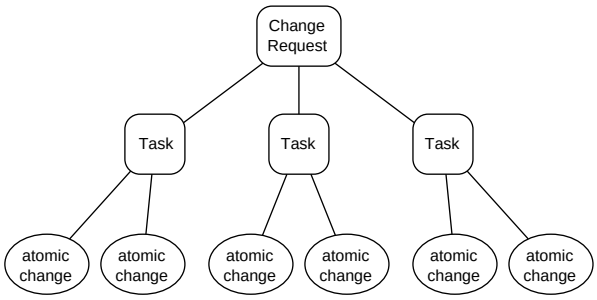

Fig. 4: Types of change

The framework in [11] also associates with each change item a reference to two documents that record change requests and changes: a *system incident report (SIR)* and a *program amendment description (PAD)*. A SIR describes what caused a change, such as the detection of a defect, a demand for improvement, or the need to adapt to a changed environment. A PAD describes the stepwise resolution of the incident that caused the change. Each SIR and each PAD is usually associated with several changes of artefacts, possibly across different subsystems. For the purposes of this paper, however, it is sufficient to simply record the task associated with a change item, since the task description in CCM contains references to associated SIRs and PADs.

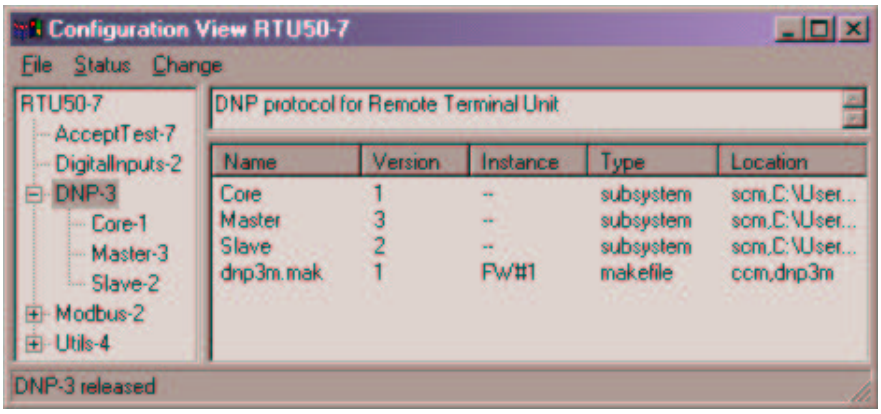

Fig. 5: A configuration view

3 The tool

The SubCM Tool is intended to support subsystem CM by implementing and specializing the framework from Section 2. The tool maintains, for each subsystem release, the SCS and the change description. Both can be viewed and navigated in the two main views of the tool: the SCS is displayed in the *configuration view* and the change description is displayed in the *change view*. These are described in more detail below, and use of the tool is discussed in Section 4.

The SubCM Tool prototype is implemented in Python with use of the graphical toolkit wxPython. It connects to CCM via CCM's command-line interface.

3.1 The configuration view

Fig. 5 shows a configuration view for version 7 of product RTU50. A configuration view has three subwindows. The left window shows a tree that represents the substructure of the opened subsystem RTU50. For each subsystem, the corresponding node in the tree displays the name and the version of the subsystem. The tree can be browsed like a filesystem tree and the two other windows show information with respect to the subsystem version that is selected in the tree, which is version 3 of DNP in Fig. 5. The right upper window shows the textual summary and the right lower window the constituents of the selected subsystem. The attributes of a constituent have been described in Section 2.1. The additional attribute *instance* is used by CCM to distinguish between different objects with the same name.

The list of constituents can be sorted by name, type or location. We distinguish three types of constituent locations: *ccm* (the constituent is stored in a CCM database), *scm* (the constituent is a subsystem and therefore stored by the SubCM Tool), and *other* (the constituent is stored elsewhere).

The status bar of the configuration view shows the current state of the selected SCS. We distinguish two states: *unreleased* and *released*. While it is being modified, the SCS is said to be unreleased. A released SCS cannot change anymore.

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

3.2 The change view

Fig. 6 shows a change view for version 4 of the subsystem DNP. The upper field shows the linear history[2] of version 4. It can be used to select two particular versions for comparison (e.g. versions 2 and 3 in Fig. 6). The corresponding change description for the selected versions appears below, with the textual summary in the middle field and the list of change items in the lower field.

For convenience, the change view records more information about change items than the change descriptions of Section 2.2. Type, instance and location information is added from the SCS, as well as a reference to the task that is associated with that change (see Section 2.3).

The list of change items can be sorted by name, change type, item type, or task. Initially, the change items are

[2]For the current version of the tool, each subsystem version is assumed to have a linear history, i.e., the tool does not currently deal with merges of different versions of a subsystem into a new version.

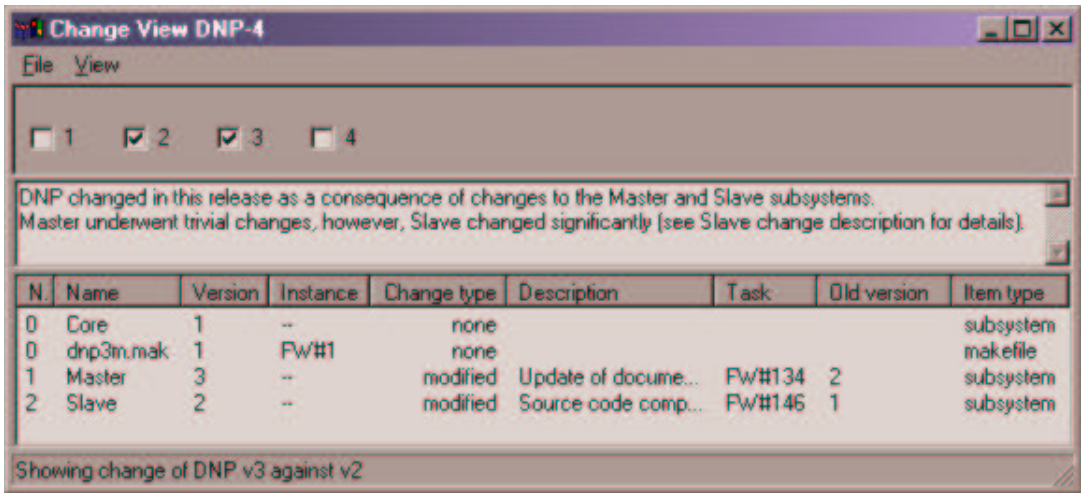

Fig. 6: A change view

listed in the order that the changes were introduced, indicated by the first column of the list. Other sorting is especially useful when viewing aggregated change descriptions, which can be lengthy: sorting by name, for example, collects all changes with respect to a particular constituent.

The change view also supports navigation. After selection of a change item that refers to a subsystem, a change view for this subsystem can be opened in a new window, which shows the change of this subsystem in more detail. Upon selection of a change item that refers to an ascii file (e.g. source code), a new window 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, which helps in verification and validation (see Section 4.4).

For reporting purposes, the SCS and the change description can be printed out from the configuration view and the change view, respectively.

4 Tool use

We now illustrate how we expect the tool would be used at various points in the lifecycle of a subsystem. This includes

1. setting up an initial SCS,
2. updating an SCS after a new subsystem build,
3. updating an SCS and change description after a product release,
4. using the tool for release integrity checks, and
5. using the tool for product support and product management.

4.1 Setting up an initial SCS

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 has to be supplied by the user. Constituents can be inserted in three ways depending on their location. Atomic CIs stored in CCM can be specified by name and the tool then presents a list of matching object versions, from which the user selects one. Subsubsystems can similarly be specified by name and the tool then presents a list of versions of the subsystem stored in the SubCM Tool, from which the user selects one. Finally, a constituent can be specified fully manually in case of constituents that are not stored in CCM or the SubCM Tool.

Often, the majority of atomic CIs of a particular subsystem are stored in CCM under one directory. The SubCM Tool facilitates automation of the setup process by allowing the user to specify the directory, and the tool then inserts all objects under that directory automatically into the SCS; this can be extended recursively to sub-directories as well.

A draft SCS can be saved, and can be edited later by adding, deleting, or modifying constituents. When the root version SCS is complete it gets *checked in* to prevent further modification. Upon check-in, the SCS changes state from *unreleased* to *released*.

4.2 Updating the SCS after a new build

The constituents of a particular subsystem continuously 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 CCM) 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. This process is now described in detail.

The SCS can be updated only if it is in the unreleased state. A released SCS can be *checked out* using the tool, which results in a new, unreleased SCS. Multiple check-outs from the same SCS result in parallel versions: i.e. *variants*. For example, the first check-out from version 5 creates version 6 and the second check-out from version 5 creates version 5.1.1 (according to CCM's numbering scheme), a variant of version 6. A corresponding change description is created for each new SCS, which is initialized with the "null" change (with change type *none*) for each constituent of the subsystem.

A subsystem configuration can be changed manually or automatically. Operations for adding, deleting and modifying (changing the version of) constituents can be invoked from the change menu of the configuration view. To invoke the delete and modify operations, the user first selects the desired constituent. In the add and modify operations, the new object version has to be specified. If that new object resides in a database then the user can name the database and the object, and the tool will then query the database and return a list of available object versions, from which the user can select one; the SCS and the change description will be updated automatically (the description and task field must be updated manually). If the object does not reside in the SubCM Tool or one of the CCM databases to which it interfaces, the version information and location must be updated manually.

The user can also invoke an automatic update of a configuration, whereupon the SubCM Tool queries CCM for all atomic changes that have been included in the latest build, and compares them with the constituents of the current configuration. If an object has changed then the tool updates the object's version number in the SCS, and adds a corresponding *modify* change item to the change description. This can be done recursively for all sub-subsystems as well.

4.3 Updating the SCS in a product release

If a product is to be released then first the build procedure in Section 4.2 is followed, which results in an update of the SCS. Various V&V procedures assure the completeness and integrity of the release. Section 4.4 describes how the tool is used for release integrity checks. If the product is ready for release then the executable and the release note should be added to the SCS. This is a good time for completing the textual descriptions of the configuration and of the changes, but this can also be done after release.

Finally, the release will be authorized and delivered, which is when the SCS should be checked in to prevent further modification of the configuration of the SCS.

4.4 Release integrity checks

An important purpose of configuration management 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. In addition, checks should be performed to ensure that functional specifications, user manuals and test specifications have been updated in accordance with the details recorded in the change request. The SubCM tool can assist with such audit tasks by allowing easy browsing of modified artefacts and comparison of changes.

The SubCM Tool 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 subsytems and may therefore be implemented by different teams. The SubCM 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 inter-subsystem 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. The SubCM Tool can assist this activity 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.

Finally, the SubCM Tool provides additional confidence that the content and characterisation of a release is correct. 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.

4.5 Product support and product management

The SubCM tool can also be used in 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 such situations, the tool allows the changes between existing and upgrade releases to be easily characterised to ensure that functionality is not lost in the upgrade or, conversely, that no undesired functionality is included.

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.

5 Conclusion

This paper introduces a framework and tool for configuration and change management of a hierarchical, "is a subsystem of"-structure on a set of software development artefacts. The design of the framework and tool was developed in collaboration with Invensys SCADA Development as a means of supporting existing CM tools and processes. Plans for integrating the tool into their current software development process have been discussed.

By providing tool support for CM of subsystems of arbitrary granularity, the following benefits are anticipated for an organisation:

- The effectiveness of the Configuration Control Board will be enhanced because of the improved visibility of changes at different levels of granularity. The SubCM Tool will also enhance coordination of changes across different developers and different development teams.
- It will be easier to develop and maintain product variants and new product families. Product artefacts can be more easily managed across multiple CM repositories or even multiple CM tools.
- The system build process can be further decentralised by assigning responsibility for different subsystems to different individuals.
- Testing effort can be more focussed by simplifying the change impact assessment as well as the integration of changes across multiple products.
- V&V processes will be improved by automating checks that sets of changes have been implemented completely and consistently across subsystems.
- The tool will aid in planning for upgrades of product releases by simplifying the comparison of previous releases with the current baseline.

In further work, we will integrate a change-tracking tool such as Telelogic's ChangeSynergy into our approach. ChangeSynergy is a web-based tool that maintains change requests and their relationship to tasks by connecting to CCM. This integration will facilitate automation of release integrity checks and product support functionality.

Acknowledgements We gratefully acknowledge the constructive comments of Paul Ellis, Alena Griffiths, Mark Staples, Jason McDonald, and Andrew Hanlon. This work was supported by the Australian Research Council, project "Automated support for verification and validation of control system software", grant No. C49937058.

References

[1] P.E. Bennett. Small modules as configuration items in certified safety critical systems. In F. Redmill and T. Anderson, editors, *Proc 6th Safety Critical Systems Symposium*, pages 62–69, Birmingham, UK, 1998. Springer Verlag.

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

[3] H.B. Christensen. Experiences with architectural software configuration management in Ragnarok. In B. Magnusson, editor, *Proc. 8th Software Configuration Management Symposium*, volume 1439 of *LNCS*, pages 67–74. Springer Verlag, 1998.

[4] 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. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, Advanced Software Development Series, pages 9–32. Research Studies Press Ltd. (John Wiley), 1994.

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

[6] CVS. Concurrent Version System. `http://www.cvshome.org`.

[7] Susan Dart. Concepts in configuration management systems. In *Proc 3rd Int Sw Config Mgmt Workshop*, pages 1–18, Trondheim, Norway, June 1991. IEEE.

[8] Susan Dart. Content change management: problems for web systems. In *Proc 9th Int System Config Mgmt Symposium (SCM-9)*, pages 1–16, Toulouse, France, Sept 1999. Springer Verlag.

[9] A. Gustavsson. *Software Configuration Management in an Integrated Environment*. PhD thesis, Department of Computer Science, Lund University, Sweden, 1990. Also available as Technical Report, LU-CS-TR:90:52.

[10] Y-J. Lin and S.P. Reiss. Configuration management with logical structures. In *Proc. IEEE 18th Conf. on Software Eng.*, pages 298–307. IEEE Press, 1996.

[11] P. Lindsay, A. MacDonald, M. Staples, and P. Strooper. A framework for subsystem-based configuration management. In D. Grant and L. Sterling, editors, *Proc. Aust. Software Eng. Conference (ASWEC 2001)*, pages 275–284. IEEE Press, 2001.

[12] P.A. Lindsay, Y. Liu, and O. Traynor. A generic model for fine-grained configuration management including version control and traceability. In *Proc. Australian Software Engineering Conference (ASWEC'97)*, pages 27–36. IEEE Press, 1997. See also SVRC TR 97-45, http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?97-45.

[13] B. Magnusson, U. Asklund, and S. Minör. Fine-grained revision control for collaborative software development. In *Proc ACM SIGSOFT'93 Symp on Foundations of Sw Eng*, Los Angeles, California, Dec 1993. ACM.

[14] Microsoft. Visual SourceSafe. `http://msdn.microsoft.com/ssafe`.

[15] Rational. ClearCase. `http://www.rational.com`.

[16] K.J. Ross and P.A. Lindsay. Maintaining consistency under changes to formal specifications. In *Proc. 1st Int. Symp. of Formal Methods Europe (FME'93)*, LNCS 670, pages 558–577. Springer Verlag, 1993. See also SVRC TR 93-3, http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?93-03.

[17] S. Sachweh and W. Schäfer. Version management for tightly integrated software engineering environments. In *Proc. 7th Int. Conf. on Software Eng Environments*, pages 21–31, The Netherlands, 1995. IEEE Press.

[18] Telelogic. Synergy toolset. `http://www.telelogic.com/products/synergy`.

[19] U.K. Ministry of Defence. Configuration management of defence material. Defence Standard 05-57/Issue 4, July 2000. `http://www.dstan.mod.uk`.

[20] B. Westfechtel. Revision control in an integrated software development environment. *ACM SIGSOFT Sw Eng Notes*, 17(7):96–105, 1989.

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