A Framework for Subsystem-based Configuration Management

Peter Lindsay* Anthony MacDonald*† Mark Staples‡ Paul Strooper*†

Abstract

Existing software Configuration Management (CM) tools are limited in the support they provide for configuration and change management of hierarchically structured software systems. This paper describes a framework for CM of subsystems *– logically coherent collections of software development artefacts, including code, documentation and test sets. The goal is to provide visibility of changes at intermediate levels between whole-system and source-code levels, thereby reducing the complexity of the build, V&V and change management processes. The framework supports characterisation of subsystems and changes to subsystems, and provides hooks into change tracking processes.*

1. Introduction

Software Configuration Management (CM) [3, 7, 22] is a key discipline for high-integrity system development [20]. CM is concerned with controlling and recording the evolution of all software development artefacts, not just source-code control. CM systems generally support a particular management philosophy that details what artefacts are put under configuration control and describes roles and responsibilities for how the management occurs [17, 18]. They incorporate system building procedures for source code, version control procedures for code and documentation, and change management procedures for all development artefacts. Also, they generally incorporate configuration status-accounting functions, which link into Verification and Validation (V&V) and Release processes [2].

This paper reports the outcomes of a pilot project that is part of an Australian Research Council SPIRT-funded collaborative research project between Foxboro Australia and the Software Verification Research Centre. Foxboro Australia build Supervisory Control and Data Acquisition (SCADA) systems, which often implement safety- and mission-critical system requirements. CM holds particular challenges for large-scale, high-assurance software developments such as these. Such developments involve configurations consisting of very large numbers of artefacts under individual version control, together with cross-artefact relationships such as traceability matrices. Because different customers have different requirements, and because systems evolve over time, many different configurations need to be supported. A Change Control Board (CCB) oversees change management, but the scale and complexity of configurations make it difficult to undertake, monitor and verify implementation of change-management action plans.

The aim of the pilot project was to increase the effectiveness of change management for complex software developments, by supporting hierarchically structured *subsystems* of a complete development. By subsystem we mean any logically coherent collection of development artefacts, including code, documentation and test sets. (The concepts introduced here can be generalised beyond software – for example, to hardware and other system documentation – but the focus of this paper is on software.) The paper describes a generic framework within which arbitrary subsystems can be put under configuration control and have their change histories recorded, and which provides hooks for tracking the cause and outcomes of changes. The framework is intended to be generic and to be implemented on top of existing CM tools.

By adding hierarchical structure to Configuration Items (CIs), the framework improves overall characterisation of system configurations, by showing clearly what versions of what components and subsystems make up a configuration. It also improves visibility of system changes, by revealing how individual subsystems have changed. We conjecture that this information will improve quality and productivity when developing code, preparing test environments, and building for release. We also conjecture that placing subsystems under configuration management will enable decentralisation of the build process. The result will be to integrate CM tools and processes more effectively into the overall system development and V&V lifecycle. As a first step in the implementation of the framework a prototype

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

†School of Computer Science and Electrical Engineering, The University of Queensland, Brisbane, Qld 4072, Australia. email: {anti, pstroop}@csee.uq.edu.au

‡Foxboro Australia, PO Box 4009, Eight Mile Plains, Qld 4113, Australia. email: markst@foxboro.com.au

0-7695-1254-2/01 $10.00 © 2001 IEEE

tool is being developed and will be trialed at Foxboro.

The paper is structured as follows: Section 2 describes CM in more detail and describes related work. Section 3 defines requirements for the framework. Section 4 defines the framework's notion of subsystem and describes how subsystems are characterised. Section 5 explores subsystem change histories, and Section 6 explores change tracking. The paper concludes with a brief summary of the framework and a discussion of implementation issues.

2. Background and related work

The key issues in establishing a CM system are:

- configuration identification – what Configuration Items (CIs) are to be considered, and what versions of what artefacts make up a given CI;
- configuration control – how are changes made to CIs, who authorises them, and how are versions of CIs stored and retrieved; and
- configuration status accounting – what is the status of a given CI (roughly, what is the development-lifecycle state of this particular version).

For our purposes, a *CM framework* describes the different types of CIs under consideration and the relationships between them. A CM framework is thus an important part of a CM Plan [18], but stops short of defining roles, responsibilities and processes for a full CM system. (Status accounting is not addressed in this paper.)

The problem of handling Configuration Items (CIs) at arbitrary levels of granularity has been recognised for some time [1, 22]. Existing CM tools [6, 13, 14, 19] often allow aggregation of artefacts into "projects", which can be treated as subsystems. These tools typically do not support versioning of aggregates nor tracking of change histories; moreover, aggregation tends to be supported only along the lines of directories (folders) in the file store of the CM repository. Previous Software Verification Research Centre (SVRC) research explored management of fine-grained development artefacts and links between them [11, 15] 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 configuration management, whereby source-code functions and classes are put under version control. Their paper includes an interesting discussion of version control issues for hierarchical systems but stops short of a full 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 [5] describes an approach to configuration and version control of software artefacts with structural and dependency links.

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 Software Engineering Environments (SEE) [9, 12, 16, 21]. 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 impact on existing CM practices.

3. Requirements for a CM framework for subsystems

This paper investigates configuration management applied to subsystems. A subsystem is a partitioning of a software development environment, which includes all artefacts used and developed within the complete software lifecycle. This partitioning enables the complete system to be viewed as a hierarchical collection of subsystems, rather than as a collection of files.

The top-level subsystem in a system is referred to as the *root* subsystem, the lowest level subsystems are called *leaf* subsystems, and subsystems that fall between these two extremes are referred to as intermediate-level subsystems, or simply *intermediate* subsystems. There may be many levels in a subsystem hierarchy and subsystems may be shared; see Figure 1.

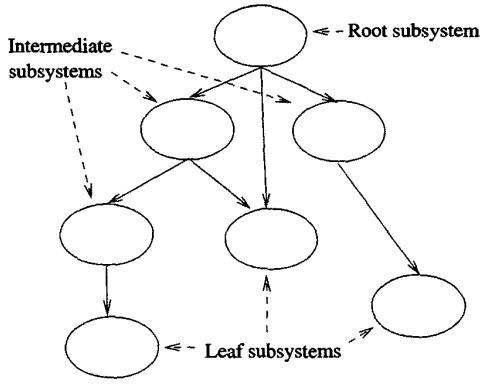

Figure 1. Terminology in a hierarchical system

Top-level subsystems are generally products delivered to the customer. Intermediate subsystems can also be products, for example communications protocols and product-

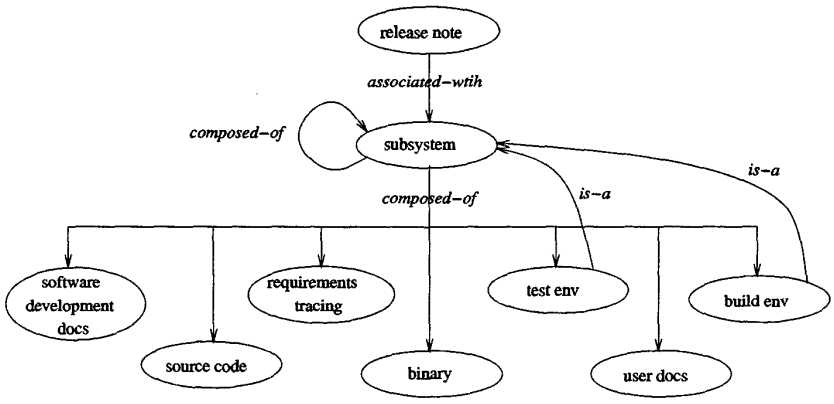

Figure 2. Entity relationships in a subsystem

support tools, such as system-diagnostic tools and tools for configuring system parameters. Leaf subsystems could be software modules which are for internal purposes only and may not be visible to customers.

In developing a configuration management framework, we need to address the following issues:

- characterisation of contents of subsystems,
- visibility of changes between subsystem releases,
- availability of defect fixes, especially tracking patches across multiple versions, and
- integration of changes to subsystems (e.g., propagation of a defect fix from an earlier version of the subsystem to the current baseline, and merging of changes from parallel variants).

A framework for subsystem-based configuration management is introduced below. The framework will enable the characterisation of a subsystem, the characterisation of change to a subsystem and the tracking of the causes and consequences of change. The framework considers the following issues:

- Characterisation - What is, and what is contained in, a subsystem?
- Change Descriptions - In what ways has a subsystem (and its components) changed between releases, and how is the change history recorded?
- Change Tracking - What causes a change to a subsystem and what relationships exist between the record of the cause, the record of the solution, and subsystems?

Note that the CM framework is concerned with the logical constitution of subsystems only, and that physical storage issues and tool support are "implementation" issues to be left for later investigation. Similarly, process issues that build on the framework will be left for future work.

4. Hierarchical structuring and characterisation of subsystems

This section describes how subsystem configurations are structured and described within the framework. The definitions are recursive, to support hierarchies of arbitrary depth. There are two aspects to system characterisation:

- hierarchical structuring of (logical) partitioning of system into configuration items, and
- identification of the particular versions of configuration items making up a given version of the system.

These are treated in Sections 4.1 and 4.2 respectively.

4.1. Subsystem configuration items

This section will identify the configuration items either found within or associated with a subsystem (as shown in Figure 2) and at which levels of a system certain items are more likely to be found. Figure 2 concerns the hierarchical structure of the (logical) containment relationship. Some configuration items are optional at certain levels, but compulsory at others, while a small subset of configuration items are compulsory at all levels within a system because they are necessary to support the CM process. The configuration items in a subsystem will be called its constituent configuration items, or *constituents* for short:

- Software development documents - Software development documents include requirements, specification, and architecture/design documents. Requirements and specification documents will usually only be found

in the root subsystem and in the subsystems corresponding to stand-alone components, such as product-support tools. They will not usually be present for subsystems that exist only as a consequence of design partitioning. Design documents should be present for all subsystems and should contain the appropriate level of detail. For example, the design document for the root subsystem should give an architectural overview of the system and comment briefly on the subsystems that are combined to make up the system. The detailed documentation on those lower-level subsystems should be within the subsystems themselves.

- Source files - Source files are usually contained in the leaf subsystems, but may be found in higher-level subsystems when code is needed for integration. Source files are the most common configuration items in traditional configuration management systems, but in a subsystem-based approach they are actually optional in some subsystems.

- Binaries - Technically, binaries should be able to be recreated, but storing binaries allows for an exact copy of the released system to be archived. For the root subsystem a binary might be an executable, and for an intermediate or leaf sub-system it might be a library.

- User documents - User documentation should be provided for any subsystem that the user can interact with. This obviously applies to the root subsystem and to any product-support tools, but also applies to any subsystem that needs to be understood by the user.

- Requirements tracing - Any requirements tracing to or from constituents of a subsystem should be provided.

- Subsystems - Subsystems can contain subsystems.

- Test environment - Testing documentation, implementation, and data files, including test plans, test drivers, test stubs, test cases, and test results are found at each level and make up the test environment. The test environment for a subsystem may support module, integration, or system testing and in fact, the test environment may support more than one type of testing. In many cases the test environment for a subsystem will itself be highly structured (e.g. with stubs for modules that interface with the subsystem) and so will itself be a subsystem.

- Build environment - Makefiles and associated configuration-specific build information, including details on third party tools (such as compilers), can be found at each level. If the subsystem can be represented by a binary of some form, whether an object file, a library or an executable, then all artefacts needed to recreate the binary should be part of the subsystem.

It may also be desirable to store data on operational usage against subsystems.

In the case of subsystems that are released to customers or other development groups, the released version is normally accompanied by a *Release Note* which describes the new version and how it should be installed. We have chosen to treat Release Notes as artifacts of the CM process rather than as constituents of the subsystem itself.

4.2. Subsystem configuration specification

This section describes how subsystems are described within our framework.

At this point, we introduce a running example to illustrate the concepts. To make the example realistic, we have based it on an abstract view of existing Foxboro software. We will use the names of the subsystems and CIs used at Foxboro, but there is no need to understand what these names stand for in order to understand the examples (DNP and Modbus are different communications protocols). The example contains

- 5 subsystems: Core, DNPMaster, DNPSlave, ModbusMaster, and ModbusSlave; and

- 2 root systems (for customers interfacing with equipment using different protocols, say): RTU+DNP and RTU+Modbus. RTU+DNP is composed of Core, DNPMaster, and DNPSlave; and RTU+Modbus is composed of Core, ModbusMaster, and ModbusSlave.

Characterising a subsystem configuration requires consideration of the constituents of the subsystem, as well as versioning and location information. Different versions of a subsystem may contain items that are either unique to the version or shared with other versions. As an example (see Figure 3), suppose that RTU+DNP (version 1) contains Core (v1), DNPMaster (v1), and DNPSlave (v1) while RTU+DNP (v2) contains Core (v1), DNPMaster (v2), and DNPSlave (v2). Furthermore, while we do not consider CM tools in detail, certain considerations for tool use cannot be ignored. In particular, all the configuration items are typically stored in one or more CM databases, and we record this as the location of the configuration item.

Full characterisation of a subsystem requires that for each constituent configuration item, the item name, version, and location are stored. This collection of identifiers is called the *Subsystem Configuration Specification* (SCS).

The SCS associated with a root system is sometimes called a Bill of Materials. In our framework, however, SCSs can be used to document individual subsystems, not just complete systems as shown in the example. Let us assume

Name: RTU+DNP
Version: v1
Description: Remote Terminal Unit using DNP protocol

Constituents	Version	Location
Core	v1	SrcCode
DNPMaster	v1	SrcCode
DNPSlave	v1	SrcCode

Name: RTU+DNP
Version: v2
Description: Remote Terminal Unit using DNP protocol

Constituents	Version	Location
Core	v1	SrcCode
DNPMaster	v2	SrcCode
DNPSlave	v2	SrcCode

Figure 3. Subsystem Configuration Specifications for two versions of RTU+DNP.

that the subsystem DNPMaster contains the following configuration items: 3 source code files, a Makefile, a protocol specification, a design document, and a test subsystem. The SCS for DNPMaster is shown in Figure 4.

Name: DNPMaster
Version: v1
Description: Master Unit part of DNP protocol.

Constituents	Version	Location
src-code1	v1	SrcCode
src-code2	v1	SrcCode
src-code3	v1	SrcCode
Makefile	v1	BuildEnv
protocol-spec	v4	Documentation
dnp-design	v3	Documentation
DnpTest	v1	TestEnv

Figure 4. SCS for DNPMaster.

For purposes of status accounting, it is desirable that SCSs be maintained throughout development, not just at release. To reduce the likelihood of human error, SCS maintenance should be as automated as possible. Section 7.2 outlines our plans for how this would be done.

5. Characterising how subsystems have changed

The changes on a subsystem can be understood by investigating the possible changes that items in a subsystem undergo, and which of the complete set of changes apply to a particular item. We consider here the logical principles for making change between subsystem releases visible. The main principle is that current and previously released versions of subsystems should be retrievable. Not only do we wish to understand how a particular release has changed from its parent, but we wish to be able to find the difference between any two releases.

5.1. Change types

Any constituent item in a subsystem can undergo one or more of the following changes between subsystem releases:

- **add**: a new item is added to the subsystem
- **delete**: an existing item is deleted from the subsystem
- **modify**: an existing item is modified (i.e., replaced by a different version of the item)
- **split**: an existing item is split into several new items (like a **delete** followed by several **adds**). For example, the subsystem Core from our example could be **split** into two subsystems, IO and Processing.
- **combine**: several existing items are combined to form a new item (like several **deletes** followed by an **add**)
- **derive**: a new item is derived from existing items (like an **add**)
- **replace**: a new item replaces an existing item (like a **delete** followed by an **add**)
- **move**: an item is moved from one subsystem to another

This list has been compiled from the kinds of change types supported by a whole range of tools (including requirements management tools) and not simply existing CM tools [11]. The first three change types are primitives; the others can be expressed as combinations of these, but are useful for understanding the evolution of a subsystem. It is our intention for many of these changes to be tool-tracked and not manually recorded: this is discussed further below.

Not all artefacts undergo all of these types of change. For example, binaries are only ever derived (through compilation): it is necessary to track the version of binaries associated with a subsystem, since they may not get updated.

5.2. Change descriptions

When a constituent of a subsystem changes, the change needs to be noted against both the constituent and the subsystem itself. This situation is analogous to source files under CM, whereby a description of recent changes are included in the file header as well as being recorded in the CM tool.

We propose a document called a *Change Description* to record the changes made to subsystems. The Change Description will contain:

- current and previous release identifiers,
- summary of changes to the subsystem as a whole, and
- a list of constituents of the subsystem and how they changed.

Note that "root versions" (initial baselines) are special cases and do not require Change Descriptions, only SCSs.

The summary of the changes should be an abstract description of the changes made to the subsystem. This description may include whether the change was corrective, perfective, adaptive, or a combination of these:

- *Corrective* changes are those made when fixing a defect in the system.
- *Perfective* changes are those made when improving a working system, such as when re-engineering the system to improve its performance or maintainability.
- *Adaptive* changes are those made when reacting to a change in the way the system will be used, such as when adapting to a new platform or adding a new communication protocol.

Furthermore, the types of configuration items affected and the consequences of the changes should be recorded. For instance, did the change predominately affect source files (as is often the case with corrective changes)? Was it an adaptive change that caused change across much of the subsystem? Or did it perhaps mainly affect the design documentation? Furthermore, the consequences of a subsystem change can either be localised (i.e., the subsystem's interface and specified behaviour do not change) or may need propoagating beyond the subsystem.

5.3. Structure of subsystem change descriptions

For our purposes, it is sufficient for a Change Description to record the current version number of the subsystem and its parent version number, together with a list of all constituents of the two versions and how they have changed: the constituent version information can be recovered from the SCSs. See Figures 5 and 6 below for examples.

We use the following labels corresponding to the change types of the previous section:

- **added**: the item did not exist in the parent version but has been added to the current version
- **deleted**: the item appears in the parent version but not the current version
- **none**: the item was unchanged between subsystem releases (this is needed for completeness)
- **split-up**: the item was split up into new items and no longer exists in the current version.
- **split-from**: the item arose from splitting up an item in the parent version
- **combined-into** and **combined-from**: similar
- **replaced** and **replacement**: similar
- **derived**: the item was derived from other items in the current version
- **moved-here**: the item was moved (unchanged) from another subsystem
- **moved-away**: the item was moved to another subsystem

Using the running example, a Change Description for RTU+DNP could be as shown in Figure 5. Continuing the example, a Change Description for DNPMaster could be as shown in Figure 6. Space does not permit illustration of the Change Description for DNPSlave.

6. Change tracking for subsystems

6.1. Motivation

In this section the focus is on (defect and) *change tracking*: i.e., the recording and tracking of problems, how they are solved, and the resulting changes they cause to subsystems. Following Foxboro's current practice, we will assume that a document called a *System Incident Report* (SIR) is used to record causes of problems, and a *Program Amendment Description* (PAD) to record the changes resulting from solving problems.

A method of relating SIRs to PADs, SIRs to releases, and PADs to releases is necessary to support effective configuration management (see Figure 7). In particular, capturing these relationships will enable later defect fixes to be more

Name: RTU+DNP
Current version: 2.0
Parent version: 1.0
Summary of changes: RTU+DNP changed in this release as a consequence of changes to both of the DNP subsystems. DNPMaster underwent trivial changes, however DNPSlave changed significantly (see DNPSlave change description for details).

Item	Change	Description
Core	none	
DNPMaster	modified	Supporting documentation within the subsystem was updated and made consistent
DNPSlave	modified	The source code was completely replaced in this release of DNPSlave

Figure 5. Example Change Description for RTU+DNP

Name: DNPMaster
Current version: 2.0
Parent version: 1.0
Summary of changes: This release differs from its parent in documentation only. The design documentation has changed and some requirements tracing has been added.

Item	Change	Description
src-code1	none	
src-code2	none	
src-code3	none	
Makefile	none	
protocol-spec	none	
dnp-design	replaced	replaced by dnp-master-design
dnp-master-design	replacement	The design document, dnp-master-design, is a new design document that accurately reflects the implemented system
dnp-trace	added	A requirements trace between the protocol-spec and the dnp-master-design has been added to the subsystem.
DnpTest	none	

Figure 6. Example Change Description for DNPMaster

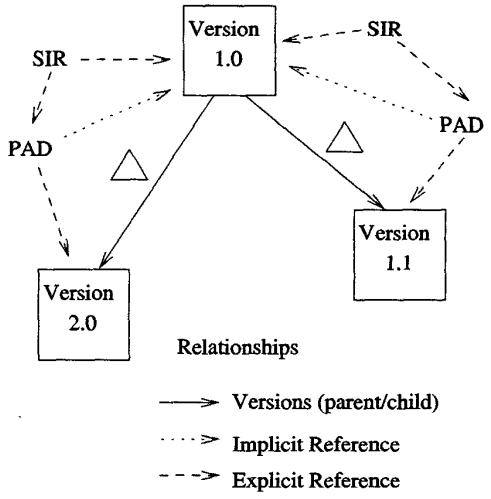

Figure 7. Relationships between SIRs, PADS, and releases

effective as the current state of a release can be better understood and it will be easier to apply a patch to a parallel or child version.

At Foxboro a SIR acts both as the trigger for action (remedial or otherwise) and as documentation of the cause (incident or creative, e.g., a bug report or requirement change). A completed SIR also references the PAD(s) that document the solution. A PAD documents the solution or why the program was amended, which files are changed, the final version, and any associated PADs.

Both of these concepts, which can be applied within subsystem-based CM, focus on the system as a whole. Each of them (SIR and PAD) focus at a fine-grained level on the cause and specifics of the change, rather than the difference between two versions, the delta (Δ). The delta is the collection of changes made to a subsystem that, when applied in sequence, move a subsystem from one version to another. Furthermore, while a single PAD may record which versions it maps between, more than one PAD can apply to a single SIR and more than one SIR can apply to a particular version.

6.2. Subsystem-based change tracking

In subsystem-based CM, associated with each subsystem should be a record of which SIRs led to the new version and which PADs document the details of that transition. Storing these relationships with a subsystem means that from any version of any subsystem it will be possible to trace not only all incidents that have been handled, but also how they were

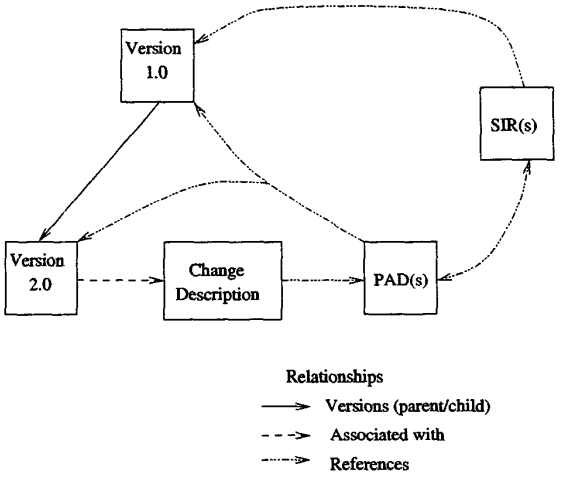

Figure 8. Relationships between Change Descriptions, SIRs, PADS, and releases

handled. This is particularly useful when trying to maintain consistency across variants of a subsystem.

To support change tracking, the last column in a Change Description contains the PADs associated with a changed subsystem. Figure 8 shows how change descriptions relate to the existing change tracking mechanisms (PADs and SIRs).

7. Conclusion

7.1. Summary

A framework for subsystem-based configuration management was introduced that enables hierarchical software subsystems, to be put under configuration and version control. By subsystems we mean logically coherent collections of software development artefacts, including code, documentation and test sets. The framework is designed to sit on top of existing CM tools and to support configuration management of subsystems independently of the tools' underlying file structures.

The framework addresses the following issues for subsystems:

- Characterisation: What is, and what is contained in, a subsystem? Subsystem Configuration Specifications (SCSs) were introduced in Section 4 to provide a mechanism for characterising a version of a subsystem.
- Change: In what ways can a subsystem (and its components) change? Changes to a subsystem were discussed in Section 5, where Change Descriptions (CDs) were introduced as a mechanism for documenting the changes between a version of a subsystem and its parent.
- Change Tracking: What causes a change to a subsystem and what relationships exist between the record of the cause, the record of the solution, and subsystems? Section 6 presented an overview of existing Foxboro change tracking mechanisms and highlighted how a simple extension to the previously introduced Change Descriptions could support change tracking across subsystems.

Figure 9 presents, in diagrammatic form, the relationships between the various entities considered in this paper. A SCS specifies a single version of a subsystem and that SCS is used to build the CD for that version of the subsystem. The CD contains a reference to the parent version as well as the current version of the subsystem. All PADs that triggered change are recorded in the CD.

7.2. Future work

Development of tool support for the approach is underway, focussing on two main areas. The first area is concerned with specifying, designing and prototyping a tool to support subsystem build management. The tool would be used by a Subsystem Build Manager to collect together the artefacts that make up a subsystem release and to generate the SCS for the new release. It will provide an SCS-like interface for the Subsystem Build Manager to interact with existing CM tools. The prototype tool will interact with Continuus ChangeSynergy [19], a CM tool from Telelogic.

At this stage no change is proposed to the way software developers use CM tools: they will continue to check in versions of basic software artefacts (source files, software development documents, binaries and user documents). However, the new tool will support the subsystem build manager by:

- retrieving version information for individual (basic) software artefacts in the tool repositories;
- managing the subsystem hierarchy;
- tracking which (versions of) artefacts have been included in the new build; and
- recording an audit trail of user operations and automatically generating the skeleton of the Change Description (CD) for the new build.

The description field of the CD will need explicit user input, but the tool would generate the item list and change types automatically.

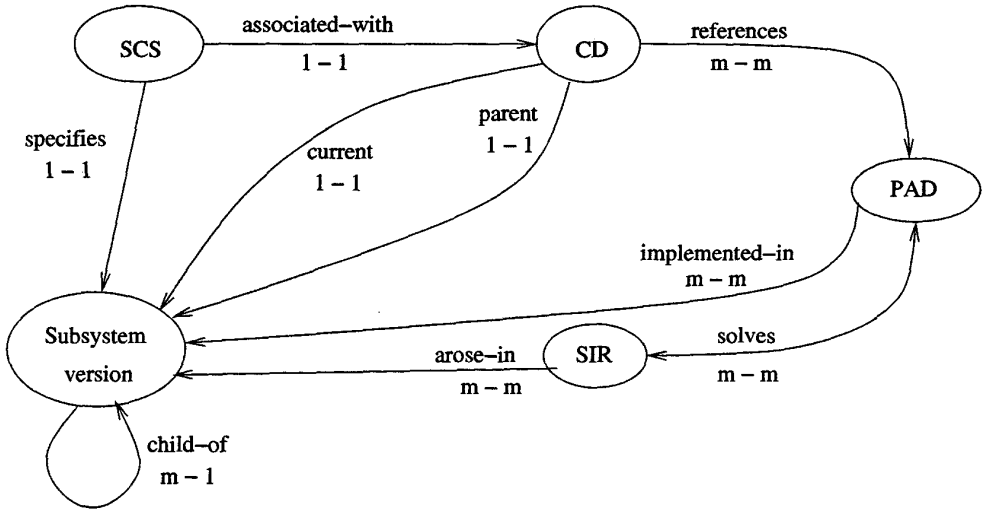

Figure 9. Entity relationships in CM framework

The second area for investigation of tool support concerns the auto-extraction of (parts of) the description field of CDs from artefact change-history information stored in the CM tool. This may involve developing guidelines for how such information should be recorded in the CM tool, and reverse engineering of a sample population in the CM tool to investigate feasibility. An outstanding research problem is to determine how far up the subsystem hierarchy change descriptions will need to percolate.

Further investigation will also be undertaken to explore addition of status accounting/auditing to the framework defined here: the goal is to support correctness checking of configurations [4]. This is expected to be mostly a matter of adding attributes to subsystem constituents to record their development/V&V status and to record who instigated the change; it will however require investigation of process changes resulting from use of subsystems and imposition of status-related constraints and checks. For example, if the subsystem contains any "derived" artefacts and subsequently undergoes change, there should be a check that the items from which the artefact was derived have not themselves changed, otherwise the derivation may no longer be valid. The addition of status accounting/auditing is our preferred solution to the problem of unauthorised change.

References

[1] P. 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. Bersoff, V. Henderson, and S. Siegel. Software configuration management: a tutorial. *IEEE Computer*, pages 6–14, Jan. 1979.

[3] E. Bersoff, V. Henderson, and S. Siegel. *Software Configuration Management.* Prentice Hall, 1980.

[4] S. Choi and W. Scacchi. Assuring the correctness of configured software descriptions. *ACM SIGSOFT Software Engineering Notes*, 14:66–75, 1989. Proc 2nd Int Workshop on Software Configuration Management.

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

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

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

[8] S. 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. Reiss. Configuration management with logical structures. In *Proc. IEEE 18th Conf. on Software Eng.*, pages 298–307. IEEE Press, 1996.

[11] P. 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 Computer Society Press, 1997. Also appears as SVRC TR 97-45, http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?97-45.

[12] 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.

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

[14] Rational. ClearCase. http://www.rational.com.

[15] K. Ross and P. 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. Also appears as SVRC TR 93-3, http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?93-03.

[16] 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 Computer Society Press.

[17] Standards Australia. Software configuration management. Australian Standard, AS 4043-1992, IEEE 1042:1987, 1992.

[18] Standards Australia. Software configuration management plans. Australian Standard, AS 4042-1992, IEEE 828:1990, 1992.

[19] Telelogic. Continuus ChangeSynergy. http://www.continuus.com.

[20] U.K. Ministry of Defence. Configuration management. Defence Standard 05-57/Issue 3, July 1993.

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

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