SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 98-10

Supporting Fine-grained Traceability in Software Development Environments

Peter Lindsay and Owen Traynor

July 1998

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

This report is an expanded version of the paper which appeared in B. Magnusson (ed), *Proceedings 8th International Symposium on System Configuration Management*, Brussels, Springer Verlag LNCS 1439, July 1998, pages 133–139.

Supporting Fine-grained Traceability in Software Development Environments

Peter Lindsay and Owen Traynor

Abstract

This paper describes the facilities currently available to support auditing and traceability within a system which provides fine-grained configuration and version management. We contend that the relationship between the configuration management system and the underlying version control system is a critical factor which governs many aspects of the facilities supporting traceability. The model of traceability is formally specified relative to our configuration and versioning models.

1 Introduction

Managing and controlling change is a critical part of software engineering. Software components typically pass through many different versions during both the initial development of a system and the ongoing maintenance of the system once deployed. The facilities that are available in such systems for tracing the evolution of requirements through the design, coding, validation and verification stages are especially critical. Such facilities are particularly important in the maintenance phase and, where conformance to standards is required, in demonstrating that such standards have been met. Conformance to such standards is not only a requirement of the initial system development, but also an ongoing requirement throughout the lifetime and evolution of a software system.

The history that documents the evolution of a software system is, in essence, the embodiment of that system. We believe that an accurate account of that history is critical in assessing the worth of the deployed system and in ensuring that subsequent developments of that system are made in a coherent and consistent fashion [2]. The history of a system also provides a wealth of information regarding design decisions and implementation choices [10, 15]. Access to a clear and consise account of such information may go a long way to reducing the effort required to rework or redevelop systems in the context of changes to requirements. As a mangement tool, this information provides valuable feedback regarding the design choices made, and processes followed, in the construction of a system.

We believe that traceability facilities that allow the documentation of a system at a finer level of granularity than the tradition build, or baseline models, provide access to essential information that is often lost in these traditional approaches. Such facilities also provide the information needed to reduce the effort required to rework or redevelop systems in the context of changes to requirements.

1.1 Software Configuration Management (SCM)

As well as providing the framework within which developers work to construct consistent system "builds", SCM provides the mechanisms needed to demonstrate traceability between the built system, the design, the requirements documents, and other tools and artifacts of a development (such as compilers and test reports). SCM provides the means for recording and controlling the "configuration" of versions of documents associated with software development, including inter- and intra-document dependencies. Regulatory and standards authorities have long recognised the importance of reliable SCM mechanisms, especially in the development of high integrity software systems [3, 8, 9].

An important supporting technology for SCM is *version control*, which concerns storage and retrieval of different versions of development components. Most version control systems attempt to maintain a record of the changes ("deltas") between different versions of components. This provides the basis for tracing the evolution of a system through its lifetime. The majority of software development companies currently use version control facilities such as RCS [16] or SCCS [11] to manage their documents and software, but such facilities operate at an inadequately coarse level of granularity (typically, whole documents or whole modules) and fall far short of users' desires.

In defining a coherent framework within which we can provide useful traceability functions, we will see that the core support for SCM (including version control) has substantial impact of the amount of effort needed to implement traceability support. We will focus on a simple example to illustrate these issues (a document conformance system) and illustrate the benefits accrued from our configuration management models from the traceability viewpoint.

1.2 Configuration Management (CM) for Formal Methods

Formal Methods of software development have particular needs in relation to CM. Formal Methods are based on the use of mathematically precise definitions of development components and their relationships, together with the use of mathematical analysis techniques – including theorem proving – for establishing correctness. The fact that individual development components have mathematical meaning makes it possible to formally verify that desired relationships hold within and between development components. In contrast to traditional development methods, cross-development configuration consistency can be defined precisely and at fine levels of granularity [13].

Consistent with this observation is our view of traceability. Our traceability model allows us to track, at fine levels of granularity, the changes that a system has undergone that moves the system through its evolving, consistent, versions. Since we are working in a context where relationship and dependencies are formally modelled, we can use this as a basis for defining formal models of traceability. Such an approach offers a great deal in the context of high integrity system development. As well as being able to trace the development of systems, we would like to be able to trace (in isolation as far as possible) the development history of individual fine-grained artifacts. In critical system development it is important to be able to trace the evolution of individual safety requirements right through the design to final implementation [9].

A very substantial side effect of our model is that of the traceability framework we propose, is that, as well as providing facilities to trace the evolution of the development of a system in terms of the individual paragraphs of a requirements document, or the declarations of a formal specification, it may be used to assess the impact of a change to a given requirement. Having a detailed history of the development of an individual requirement, for example, also provides a basis for assessing the impact of changing that requirement.

The ARC-funded *Fine-Grained Configuration Management (FGCM)* project at the SVRC is establishing a framework for fine-grained configuration management. The framework builds on a programme of work carried out by PhD student Kelvin Ross under the supervision of the first author, investigating the application of SCM techniques to formal development [12, 13, 14]. The aim of the framework is to allow developers to support their correctness claims with evidence that, not only have the individual components of a system been developed correctly, but that the combination and integration of these components has been done in a consistent manner and that the final result is derived from consistent, complete and up-to-date development components. The framework is intended to apply not only to specifications, designs and programs, but also to fine-grained development components such as the specification components, reviews, change requests, refinements, design decisions, test sets, theories and proofs that are generated as part of the development process [6].

1.3 High integrity software engineering

We consider the definition of a coherent framework, within which configuration and version management can be carried out, as an important prerequisite in the development of trusted and cost-effective environments for the development of critical software. The processes that define the development activities in such trusted environments must be based on sound underlying technology and models that allow the impact of any development step (in terms of the consistency of the relationships between the underlying artifacts) to be accurately assessed. Existing Software Engineering Environments (SEEs) use relatively untrusted standard version-management technology in the development of critical systems; this is clearly a weak link since these technologies have no coherent formal basis for consistency checking.

The need for careful control of the development process must be balanced against the need for flexibility. Users will not accept a development process that is overly constraining. Similarly, it is vitally important for encouraging industrial uptake that Formal Methods be adaptable to different situations, project structures and so on, without sacrificing the trustworthiness of the environments.

Our approach has been to define configuration consistency models (or *configuration models*, for short) which define the key configuration items, the relationships between them, and the consistency and completeness conditions desired for the configuration. In our approach, configuration models would form the core part of SEEs, with development processes defined relative to the core models. This means that the consistency and completeness of a development could be established largely independently of the development process applied, giving flexibility and trustworthiness in the one framework [4, 5, 7].

In the spirit of these configuration models, we take a similar approach when defining the notions of traceability. Out trace models are defined relative to the core configuration models. Again, the form of this definition allows us to establish consistency criteria for our framework. Ensuring that such consistency criteria are met, is an important issue, especially in the context of high-integrity development.

1.4 This report

This report gives preliminary conclusions from a case study in adding fine-grained versioning, configuration control and traceability to a simple document conformance system. We concentrate here of presenting our results from the traceability viewpoint. Section 2 describes the general framework within which we are working and introduces the example that is used in the rest of the paper. Sections 3–7 formally specify the data model on which the framework is based. Conclusions and future work are presented in section 8.

2 The case study

2.1 Documents

We start by felling some trees to better see the forest. We shall consider part of a development consisting simply of two documents – called A and B here, for short – which consist of sets of requirements and which are expected to conform with one another in some way. For example, A might be a Software Requirements Specification (SRS) and B an Architectural Requirements Specification, describing a module structure and how the requirements from the SRS are allocated to modules: see Fig. 1.

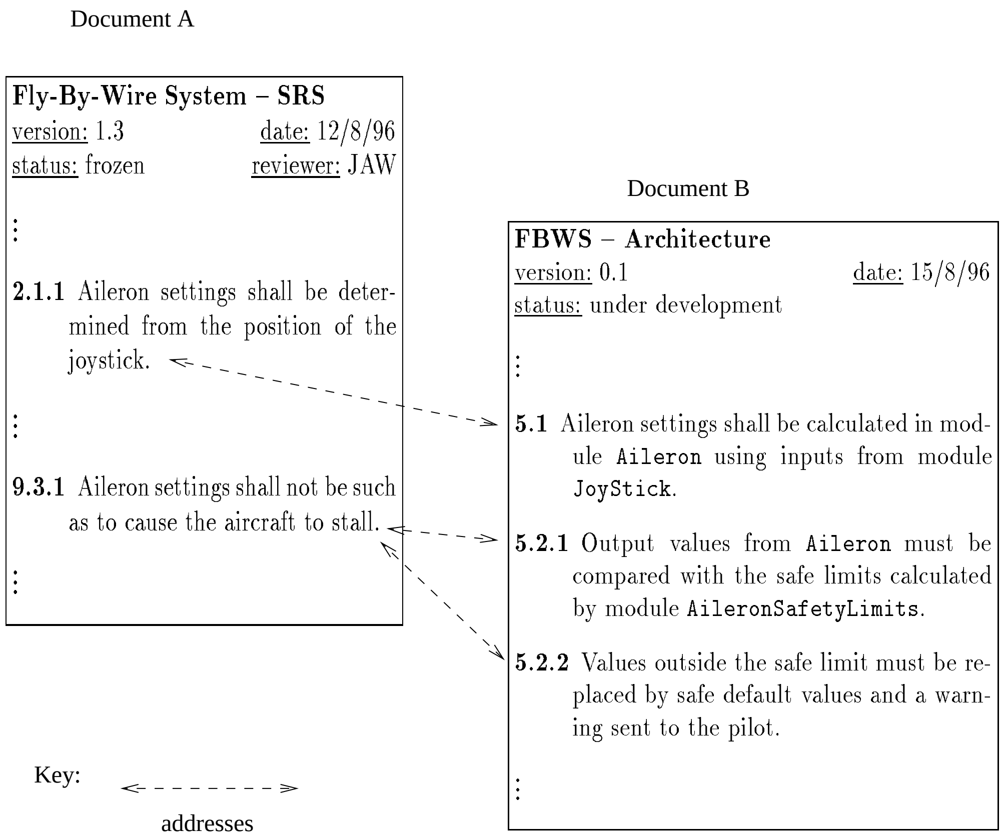

Figure 1: An example of conforming documents, illustrating how requirements in one are addressed by requirements in the other.

2.2 The dependency relation

The dependency relation to be managed is the relationship which describes how requirements in B address requirements in A. In general, such a relationship is many-to-many: e.g. a single SRS requirement may be addressed in different places in the architecture document, and a single architectural requirement may address multiple SRS requirements. The *addresses* relation is analogous in some ways to the 'is up to date with respect to' notion for traditional SCM systems, as used by the Unix MAKE facility for example. Note however that for informal objects such as requirements, the relation is user-determined and cannot be automated.

A *conformance matrix* (or traceability table) is a common means of indicating the relationship between two development documents. For each item in one of the development documents, the matrix lists the corresponding items in the other document, typically by paragraph or section number. In our case, the conformance matrix will record where each individual requirement in the SRS is addressed in the architecture document: see Fig. 2.

SRS (v1.3)	Architecture (v0.1)
⋮	
2.1.1	..., 5.1,...
⋮	
9.3.1	..., 5.2.1, 5.2.2,...
⋮	

Figure 2: Part of the conformance matrix for the example above.

2.3 Version and configuration management

For simplicity, we use a simple "version tree" model as the basis for version control, for objects of all granularities: paragraphs, documents, matrics, etc. This part of the model is reasonably generic, but could easily be replaced by something more appropriate if desired.

The configuration management problem for the case study is to manage the "configuration" consisting of the two documents and their conformance matrix. We say the conformance matrix is *complete* if each of A's requirements are addressed somewhere in B. As part of the case study, we shall assume that at any time the conformance matrix may be incomplete, but it is always *correct*: i.e., that if the matrix says that r addresses t, then this relationship was determined by a user and the requirements have not changed in the interim.

It is not reasonable to expect to be able to develop generic version and configuration management functionality – especially at fine levels of granularity – because of the widely varying requirements of different applications, development methodologies, company policies, etc. However, as we shall show, it is possible to develop a reasonably generic framework in which generic definitions and functionality are supplemented and instantiated by application-specific *version and configuration management (V&CM) policies*.

Let us suppose, for the purposes of the case study, that the following V&CM policy is in operation for coarse-grained configuration items:

- all (and only) frozen versions of documents will be stored in the project archive;
- document A can be frozen at any time;
- document B can be frozen only after a review confirms that the appropriate version of A has been frozen and the conformance matrix is complete.

As the case study progresses, we shall introduce V&CM policies appropriate to the finer-grained objects involved.

2.4 Change tracking

In our model we require that modifications to (evolution of) individual requirements are carried out within a pre-defined framework. Experience suggests that the following set of change types is a useful factoring of concerns:

add: create a new paragraph with no prior history

split: create a number of new paragraphs by splitting an existing paragraph

combine: create a new paragraph by combing existing paragraphs

delete: delete a paragraph

replace: replace existing paragraphs by a new paragraph

modify: modify the paragraph without changing its meaning

We shall extend our base model for documents with information relevant to how the current version of the document has changed since its last major version. The major versions of documents define baselines relative to which these *delta lists* (or change logs) are constructed. Note that in principle the modelling of the documents themselves need not be changed: rather the versioning models that are inherited are extended with the enriched concept of change.

Fig. 3 illustrates one kind of traceability ("forwards tracing") that will be possible as a result of our approach: given a specific version (p2,v0) of a requirement, report how the requirement changed subsequently. Fig. 4 illustrates backwards tracing: given a specific version (p13,v0) of a requirement, report the evolution that resulted in the requirement.

2.5 Requirements tracing

The second major form of traceability we provide is concerned with tracking dependencies – an important part of high integrity development and one of the main mechanisms required in development audit and evaluation. There are many ways in which one might want to trace the evolution of an individual requirement through a development. For our case study, the kinds of checks one might want to apply include the following:

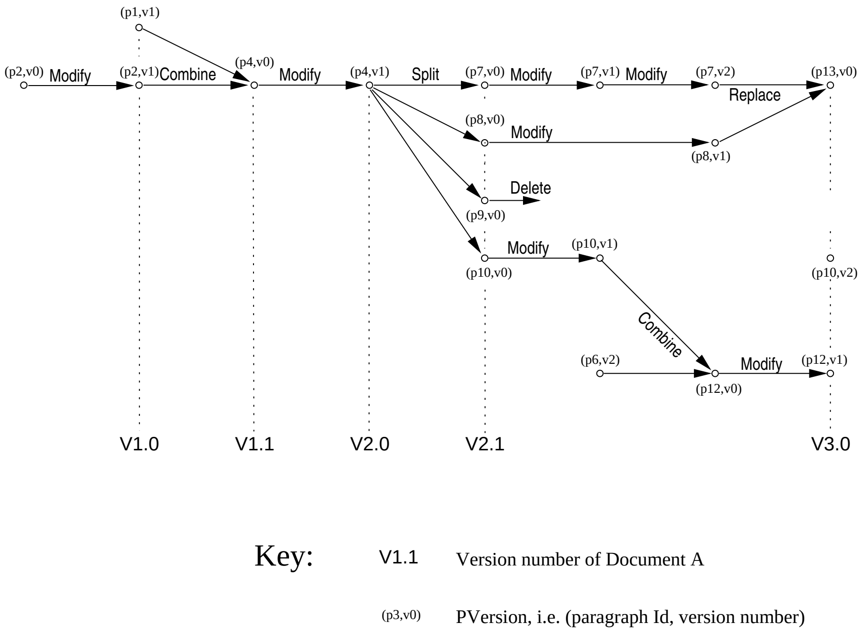

Figure 3: Forwards tracing of the evolution of a given paragraph version

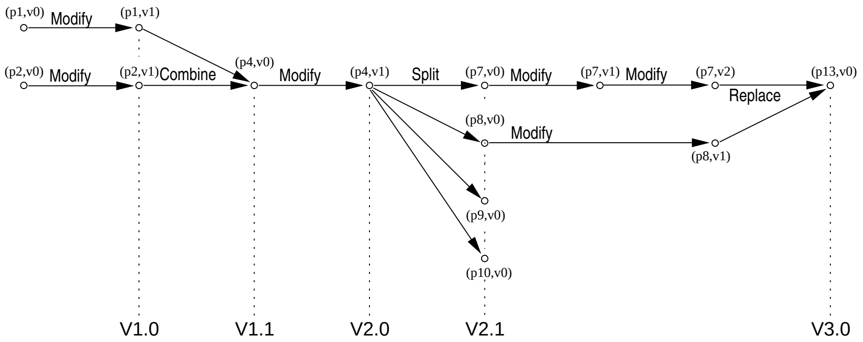

Figure 4: Backwards tracing of the evolution of a given paragraph version

- given a requirement in A, find which requirements in B address it;
- given a B requirement, find which A requirements it addresses;
- find which A requirements have not yet been addressed;
- find which B requirements are extraneous (i.e., do not address an A requirement);
- report the evolution of a given requirement (i.e., the version of the in which it originated, and how it changed in subsequent versions of the document).

3 Version control

Our simple model for version control is based on *forests* of version trees and parameterised over the type *Type* of objects being placed under version control. The attributes of version forests are defined as follows:

- Each node in a version tree is labelled with a unique *version label*, from the set *VLabel*.
- The mapping *deref* 'dereferences' version labels, yielding the content of a particular version.
- The mapping *parent* returns the (unique) parent of a non-root node. The mapping is acyclic.
- The set *frozen* contains the labels of the frozen nodes.

$VForest[Type]$

$$\begin{array}{l}
nodes : \mathbb{P}\ VLabel \\
deref : VLabel \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow Type \\
parent : VLabel \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow VLabel \\
frozen : \mathbb{P}\ VLabel \\
roots : \mathbb{P}\ VLabel \\
\hline
nodes = \mathrm{dom}\ deref \\
\mathrm{dom}\ parent \cup \mathrm{ran}\ parent \cup frozen \subseteq nodes \\
roots = nodes \setminus \mathrm{dom}\ parent \\
\forall v : VLabel \bullet (v, v) \notin parent^{+}
\end{array}$$

As a V&CM policy decision, parents will be required to be frozen:

$$\mathrm{ran}\ parent \subseteq frozen$$

4 Paragraphs

4.1 Paragraph version management

Paragraphs will be identified by a label from the type PId. A specific *paragraph version* consists of the paragraph's identifier and a specific version label:

$$PVersion == PId \times VLabel$$

The collection of *paragraph changes* is modelled as follows:

$$\begin{array}{lll} PChange & ::= & add\langle\langle PVersion\rangle\rangle \\ & | & delete\langle\langle PVersion\rangle\rangle \\ & | & split\langle\langle PVersion \times PVersion^{+}\rangle\rangle \\ & | & combine\langle\langle PVersion^{+} \times PVersion\rangle\rangle \\ & | & derive\langle\langle PVersion^{+} \times PVersion\rangle\rangle \\ & | & replace\langle\langle PVersion^{+} \times PVersion\rangle\rangle \\ & | & modify\langle\langle PId \times VLabel \times VLabel\rangle\rangle \end{array}$$

The following defines a predicate for checking whether a change affects a given paragraph version:

$$\begin{array}{l} isChangedBy : PVersion \leftrightarrow PChange \\ \hline \neg\ isChangedBy(opv, add(pv)) \\ isChangedBy(opv, delete(pv)) \Leftrightarrow opv = pv \\ isChangedBy(opv, split(pv, pvs)) \Leftrightarrow opv = pv \\ isChangedBy(opv, combine(pvs, pv)) \Leftrightarrow opv \in \operatorname{ran} pvs \\ \neg\ isChangedBy(opv, derive(pvs, pv)) \\ isChangedBy(opv, replace(pvs, pv)) \Leftrightarrow opv \in \operatorname{ran} pvs \\ isChangedBy(opv, modify(a, u, v)) \Leftrightarrow opv = (a, u) \end{array}$$

The following predicate checks whether a change creates a given paragraph:

$$\begin{array}{l} isCreatedBy : PVersion \leftrightarrow PChange \\ \hline isCreatedBy(npv, add(pv)) \Leftrightarrow npv = pv \\ \neg\ isCreatedBy(npv, delete(pv)) \\ isCreatedBy(npv, split(pv, pvs)) \Leftrightarrow npv \in \operatorname{ran} pvs \\ isCreatedBy(npv, combine(pvs, pv)) \Leftrightarrow npv = pv \\ isCreatedBy(npv, derive(pvs, pv)) \Leftrightarrow npv = pv \\ isCreatedBy(npv, replace(pvs, pv)) \Leftrightarrow npv = pv \\ isCreatedBy(npv, modify(a, u, v)) \Leftrightarrow npv = (a, v) \end{array}$$

A check that two changes don't create – or try to change – the same thing:

$$\begin{array}{l} noninterfering : PChange \leftrightarrow PChange \\ \hline noninterfering(c_1, c_2) \Leftrightarrow \\ \quad \forall pv : PVersion \bullet \\ \quad\quad \neg (isCreatedBy(pv, c_1) \land isCreatedBy(pv, c_2)) \land \\ \quad\quad \neg (isChangedBy(pv, c_1) \land isChangedBy(pv, c_2)) \end{array}$$

A *change set* is a set of noninterfering changes:

$$PChangeSet == \{cs : \mathbb{P}\, PChange \mid \forall c_1 \neq c_2 \in cs \bullet noninterfering(c_1, c_2)\}$$

Two change sets are disjoint if their elements are pairwise noninterfering:

$$\begin{array}{l} disjoint : PChangeSet \leftrightarrow PChangeSet \\ \hline disjoint(cs_1, cs_2) \Leftrightarrow \forall c_1 \in cs_1, c_2 \in cs_2 \bullet noninterfering(c_1, c_2) \end{array}$$

4.2 Paragraph collections

A collection of *specific paragraph versions*, in which each paragraph is represented at most once, is modelled as follows:

$$PCollection == PId \nrightarrow VLabel$$

The following function applies a change to a paragraph collection, if it makes sense to do so:

$$\begin{array}{l} applyChange : PCollection \times PChange \nrightarrow PCollection \\ \hline a \notin \operatorname{dom} pc \Rightarrow applyChange(pc, add(a, v)) = pc \cup \{a \mapsto v\} \\ pv \in pc \Rightarrow applyChange(pc, delete(pv)) = pc \setminus \{pv\} \\ pv \in pc \land \forall i : 1 \mathinner{..} n \bullet a_i \notin \operatorname{dom} pc \land \forall j : 1 \mathinner{..} i-1 \bullet a_j \neq a_i \\ \quad \Rightarrow applyChange(pc, split(pv, \langle (a_1, v_1), \ldots, (a_n, v_n) \rangle)) \\ \quad\quad = (pc \setminus \{pv\}) \cup \{i : 1 \mathinner{..} n \bullet a_i \mapsto v_i\} \\ \operatorname{ran} pvs \subseteq pc \land a \notin \operatorname{dom} pc \\ \quad \Rightarrow applyChange(pc, combine(pvs, (a, v))) = (pc \setminus \operatorname{ran} pvs) \cup \{a \mapsto v\} \\ \operatorname{ran} pvs \subseteq pc \land a \notin \operatorname{dom} pc \\ \quad \Rightarrow applyChange(pc, derive(pvs, (a, v))) = pc \cup \{a \mapsto v\} \\ \operatorname{ran} pvs \subseteq pc \land a \notin \operatorname{dom} pc \\ \quad \Rightarrow applyChange(pc, replace(pvs, (a, v))) = (pc \setminus \operatorname{ran} pvs) \cup \{a \mapsto v\} \\ pc(a) = u \Rightarrow applyChange(pc, modify(a, u, v)) = pc \oplus \{a \mapsto v\} \end{array}$$

The clauses in the above definition are intended to be exhaustive: i.e., $applyChange(pc, c)$ is not defined if it is not covered by one of the clauses above.

The above function extends in the obvious way to a function for applying a set of changes, if it is possible to do this in an unambiguous manner:

$$
\begin{array}{l}
applyChanges : PCollection \times PChangeSet \rightarrow PCollection \\
\hline
applyChanges(before, cs) = after \Leftrightarrow \\
\quad \exists\, cseq : \operatorname{seq} PChange;\ pcseq : \operatorname{seq} PCollection \bullet \\
\quad\quad \#pcseq = \#cseq + 1 \\
\quad\quad \operatorname{ran} cseq = cs \\
\quad\quad head\ pcseq = before \land last\ pcseq = after \\
\quad\quad \forall\, i : 1 \mathrel{..} \#cseq \bullet applyChange(pcseq(i), cseq(i)) = pcseq(i+1)
\end{array}
$$

Note that the order in which the changes are applied is not necessarilly uniquely determined; however, the result *is* required to be uniquely determined. It is difficult to define the precondition explicitly, but the requirement that the change set be noninterfering is easy to check and takes care of most problems.

4.3 The complete paragraph-history data model

The following data type specification models a complete collection of paragraph version trees and their derivation (via a set of changes). For genericity, the paragraph contents are taken to be of a given generic type *Type*. For convenience, we include an auxiliary predicate *okPVersion* for checking whether a given paragraph version is present in the collection.

$$
\begin{array}{l}
\underline{\quad ParaHistory[Type] \quad} \\
ptree : PId \twoheadrightarrow VForest[Type] \\
pchanges : PChangeSet \\
okPVersion : \mathbb{P}\, PVersion \\
\hline
okPVersion = \bigcup \{ a \in \operatorname{dom} ptree \bullet \{ v \in ptree(a).nodes \bullet (a, v) \} \} \\
\forall\, pv \in okPVersion \bullet \exists_1\, c \in pchanges \bullet isCreatedBy(pv, c)
\end{array}
$$

The invariant says that the changes are required to be pairwise noninterfering and that each paragraph version has a unique point of creation. Further properties can be added to the invariant to reflect the fact that the paragraph history has been derived in a well-defined manner.

The following V&CM policy will be applied to paragraphs:

- When paragraphs are first created, their initial versions are assigned to root nodes.
- If a change is ever applied to a paragraph, the changed version must be frozen.
- When paragraphs are modified, the new version becomes a child of the old version.

$$\forall (a,v) \in okPVersion;\ c \in pchanges \bullet$$
$$\quad isCreatedBy((a,v),c) \Rightarrow v \in ptree(a).roots$$
$$\forall (a,v) \in okPVersion;\ c \in pchanges \bullet$$
$$\quad isChangedBy((a,v),c) \Rightarrow v \in ptree(a).frozen$$
$$\forall\, a : PId;\ u, v : VLabel \bullet modify(a,u,v) \in pchanges \Rightarrow$$
$$\quad okPVersion(a,u) \wedge okPVersion(a,v) \wedge ptree.parent(v) = u$$

5 Documents

5.1 Document versions

For the purposes of this paper, each version of an evolving document is modelled as a collection of specific paragraph versions and a *delta set* of the paragraph changes that have been made since the document was last frozen. For convenience, we include an auxiliary variable *pids* representing the set of identifiers of paragraphs in the document.

Document

$$pcoll : PCollection$$
$$delta : PChangeSet$$
$$pids : \mathbb{P}\, PId$$

$$pids = \operatorname{dom} pcoll$$

5.2 The complete document-history data model

The complete history of a document consists of a complete paragraph history together with a collection of document versions: see Fig. 5.

DocHistory

$$ParaHistory[Text]$$
$$docs : VForest[Document]$$

$$\forall\, V_1 \neq V_2 \in docs.nodes \bullet disjoint(docs.deref(V_1).delta, docs.deref(V_2).delta)$$
$$pchanges = \bigcup\{V \in docs.nodes \bullet docs.deref(V).delta\}$$
$$\forall\, V \in \operatorname{dom} docs.roots \bullet D.pcoll = applyChanges(\varnothing, D.delta)$$
$$\quad \textbf{where}\ D = docs.deref(V)$$
$$\forall\, V \in \operatorname{dom} docs.parent \bullet$$
$$\quad newdoc.pcoll = applyChanges(olddoc.pcoll, newdoc.delta)$$
$$\quad \textbf{where}\ newdoc = docs.deref(V),\ olddoc = docs.deref(docs.parent(V))$$

The invariant says:

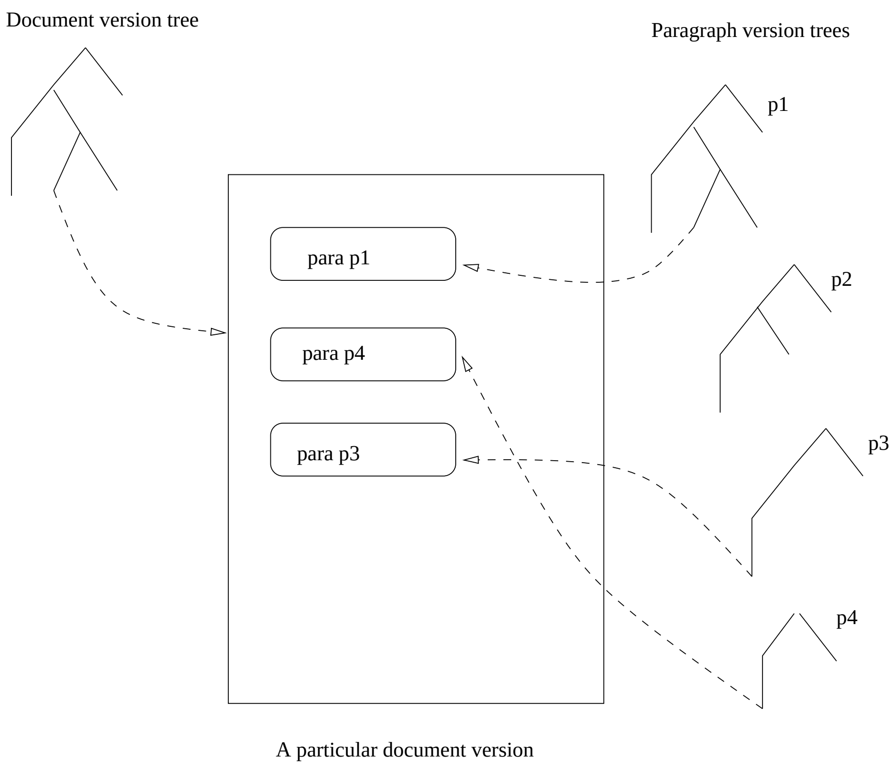

Figure 5: Each version of a document contains a specific set of paragraphs.

- The individual delta sets are all disjoint.
- The paragraph history is in step with the document history, in the sense that it has been derived from the changes recorded in the delta sets of the various document versions.
- For any root version of the document, the delta set records that version's evolution from the null (empty) document.
- For any non-root version of the document, the delta set records that version's evolution from its parent version.

The following V&CM policy will be applied to documents: when a document version is frozen, all its corresponding paragraph versions should also be frozen.

$$\forall V \in docs.frozen \bullet \forall (a, v) \in docs.deref(V).pcoll \bullet v \in ptree(a).frozen$$

6 Tracing evolution of paragraphs

This section shows that the above model is rich enough to allow the evolution of individual paragraphs to be traced (backwards and forwards) through a document's history.

6.1 Forwards tracing

The following function defines the set of paragraphs which result from a change to a given paragraph:

$$givesRiseTo : PVersion \times PChange \rightarrowtail \mathbb{P}\, PVersion$$

$$\begin{array}{l}
\operatorname{dom} givesRiseTo = isChangedBy \\
givesRiseTo(pv, delete(pv)) = \varnothing \\
givesRiseTo(pv, split(pv, pvs)) = \operatorname{ran} pvs \\
i \in 1 \mathrel{..} n \Rightarrow givesRiseTo(pvs(i), combine(pvs, pv)) = \{pv\} \\
i \in 1 \mathrel{..} n \Rightarrow givesRiseTo(pvs(i), replace(pvs, pv)) = \{pv\} \\
givesRiseTo((a, u), modify(a, u, v)) = \{(a, v)\}
\end{array}$$

(Note that the above function does not include paragraphs that are "derived" from the given paragraph, since the latter are not strictly changes to the given paragraph.)

The following function extracts each change (and corresponding document version) which occurs "downstream" in a given paragraph's evolution:

$$forwardsTrace : PVersion \times DocHistory \rightarrow\!\!\!\!\!\!+\ \mathbb{P}(PChange \times VLabel)$$

$$(pv, D) \in \operatorname{dom} forwardsTrace \Leftrightarrow pv \in D.okPVersion$$
$$forwardsTrace(pv, D) =$$
$$\quad \text{if } \exists\, V \in D.docs.nodes;\ c \in D.docs.deref(V).delta \bullet isChangedBy(pv, c)$$
$$\quad \text{then } \{(c, V)\} \cup \bigcup\{npv : givesRiseTo(pv, c) \bullet forwardsTrace(npv, D)\}$$
$$\quad \text{else } \varnothing$$

(Note that the fact that this definition is well formed depends on an unrecorded assumption about the "well foundedness" of the set of paragraph changes (*pchanges*). We should really add an appropriate property to the configuration invariant on *ParaHistory* to cover this.)

6.2 Backwards tracing

The next function defines the set of paragraphs which a given change "uses":

$$usesPVs : PChange \rightarrow\!\!\!\!\!\!+\ \mathbb{P}\, PVersion$$

$$\operatorname{dom} usesPVs = \operatorname{ran} isCreatedBy$$
$$usesPVs(add(pv)) = \varnothing$$
$$usesPVs(split(pv, pvs)) = \{pv\}$$
$$usesPVs(combine(pvs, pv)) = \operatorname{ran} pvs$$
$$usesPVs(derive(pvs, pv)) = \operatorname{ran} pvs$$
$$usesPVs(replace(pvs, pv)) = \operatorname{ran} pvs$$
$$usesPVs(modify(a, u, v)) = \{(a, u)\}$$

The next function finds the change (and corresponding document version) which created a given paragraph version:

$$creationPoint : PVersion \times DocHistory \rightarrow\!\!\!\!\!\!+\ PChange \times VLabel$$

$$(pv, D) \in \operatorname{dom} creationPoint \Leftrightarrow pv \in D.okPVersion$$
$$creationPoint(pv, D) = (c, V) \Leftrightarrow$$
$$\quad isCreatedBy(pv, c) \wedge V \in D.docs.nodes \wedge c \in D.docs.deref(V).delta$$

The following function extracts the important "creation" steps along the way to arriving at a given paragraph:

$$backwardsTrace : PVersion \times DocHistory \rightarrow\!\!\!\!\!\!+\ \mathbb{P}(PChange \times VLabel)$$

$$(pv, D) \in \operatorname{dom} backwardsTrace \Leftrightarrow pv \in D.okPVersion$$
$$backwardsTrace(pv, D) = \{(c, V)\} \cup \bigcup\{opv : usesPVs(c) \bullet backwardsTrace(opv, D)\}$$
$$\quad \textbf{where}(c, V) = creationPoint(pv, D)$$

(A similar remark about well formedness of the *forwardsTrace* definition applies here.)

7 Document conformance

The case study is completed by demonstrating how conformance between pairs of documents can be modelled.

A conformance matrix is modelled as a relation between individual paragraphs in the two documents:

$CMatrix == PId \leftrightarrow PId$

The whole configuration is modelled as a pair of document histories, a forest of conformance matrix versions, and a relation which records which versions of the three objects make up "legitimate" configurations:

$$
\begin{array}{l}
\textit{DocPair} \\
\hline
A, B : \textit{DocHistory} \\
\textit{cmatrix} : \textit{VForest}[\textit{CMatrix}] \\
\textit{corres} : \mathbb{P}(\textit{VLabel} \times \textit{VLabel} \times \textit{VLabel}) \\
\hline
\forall (V_A, V_B, V_C) \in \textit{corres} \bullet \\
\quad V_A \in A.\textit{docs}.\textit{nodes} \wedge V_B \in B.\textit{docs}.\textit{nodes} \wedge V_C \in \textit{cmatrix}.\textit{nodes} \\
\quad \forall (a, b) \in \textit{cmatrix}.\textit{deref}(V_C) \bullet \\
\qquad a \in A.\textit{docs}.\textit{deref}(V_A).\textit{pids} \wedge b \in B.\textit{docs}.\textit{deref}(V_B).\textit{pids} \\
\hline
\end{array}
$$

The *corres* relation records which versions of the three objects make up recognised configurations. The invariant says that

- there are no dangling references in the *corres* relation
- there are no dangling references to paragraph identifiers in the conformance matrices.

The V&CM policy from Section 2 will be strengthened as follows:

- The conformance matrix and B-document should be managed as a single configuration: i.e., there is a one-one correspondence between versions of the two objects.
- Moreover, each version of said configuration should refer to a single version of the A-document.
- If B is frozen then the corresponding A should also be frozen and the conformance matrix should be frozen and complete (i.e., every A-paragraph should be addressed by at least one B-paragraph).

(Note that in multidocument situations there may be a possibility of deadlock here.)

$$\forall (V_A, V_B, V_C), (V'_A, V'_B, V'_C) \in corres \bullet V_B = V'_B \Leftrightarrow V_C = V'_C$$
$$corres^\sim \in VLabel \times VLabel \pinj VLabel$$
$$\forall (V_A, V_B, V_C) \in corres \bullet V_B \in B.docs.frozen \Rightarrow$$
$$\quad V_A \in A.docs.frozen \land V_C \in cmatrix.frozen \land$$
$$\quad \mathrm{dom}\, cmatrix.deref(V_C) = A.docs.deref(V_A).pids$$

This policy could usefully be strengthened, say in COMPUSEC applications, to say that every B-paragraph should address at least one A-paragraph (e.g. to ensure that no unauthorised functionality has been added).

8 Conclusions

8.1 Summary

This paper described a case study in fine-grained version and configuration management from the perspective of providing integrated functionality to support auditing and traceability. We demonstrated that fine-grained versioning of versioned objects provides substantially more flexibility than traditional approaches that manage only high-level coarse-grained objects.

There are two dimensions to the benefits accrued by the use of fine-grained versioning models in our case study. The first is due to the fact that we consider the individual components of our systems as first class citizens in the context of configurations. This allows substantial flexibility in the way in which consistency of an overall system is determined, as well as focusing attention on the specific objects undergoing change.

The second benefit comes from the actual versioning of the system components themselves. It allows us to define consistency criteria in terms of the conditions that must be satisfied by the individual components. We can then show that the chosen versioning model actually meets the criteria.

8.2 Further Work

During our work a number of other issues arose, particularly with respect to possible extensions to our framework which would allow developers to reduce the impact of change (as opposed to accurately assessing the actual work required to react to a change).

It's clear that, by supporting more sophisticated structuring mechanisms the impact of change could be localised better. We intend to test our hypothesis that the versioning model outlined here will scale to more complex structures, to provide a basis for the flexible construction and maintenance facilities suitable for large-scale development.

The model we have presented is a core model. In addition to this, one would typically define high (user)-level processes based on these models. Whereas consistency constraints for objects within a system are defined by the model, the process which evolves and uses these underlying concepts need not be fixed. Our framework can be used as a basis upon which more sophisticated process models can be developed. Such models can offer context-sensitive guidance to the users of systems, as well as the opportunity to further constrain the way in which a system is used. It is possible to define and reason about intermediate states of configuration consistency, and to offer guidance on how to bring the system back into a consistent state, for example. We have illustrated these ideas on theory management [4, 7].

Finally, we have an ongoing effort in prototyping tools to support our fine-grained configuration management framework using object-oriented database technology [1].

8.3 Acknowledgements

The authors gratefully acknowledge the useful contributions of Yaowei Liu and Sabine Sachweh to the work presented here.

References

[1] F. Bancilhon, C. Delobel, and G. Harrus. *Building an Object-Oriented Database System: the Story of O2*. Morgan Kaufman, 1992.

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

[3] IEC. Functional Safety: Safety-Related Systems. Draft International Standard IEC 1508, June 1995.

[4] P.A. Lindsay. A formal approach to specification and verification of task management in interactive systems. *IEE Proceedings of Software Eng*, 144(4):206–214, August 1997. (Formerly Sw Eng Journal.) Also appears as SVRC TR 97-23.

[5] 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 Computer Society Press, 1997. SVRC TR 97-45.

[6] P.A. Lindsay, Y. Liu, and O. Traynor. Managing document conformance: a case study in fine-grained configuration management. *Aust Comp Sci Communications*, 19(1):373–382, 1997. Also appears as SVRC TR 96-20.

[7] P.A. Lindsay and O. Traynor. Version and configuration management of formal theories. In *Proc. Formal Methods Pacific (FMP'97)*, pages 165–185. Springer Verlag, 1997. Also appears as SVRC TR 97-13.

[8] U.K. Ministry of Defence. The Procurement of Safety Critical Software in Defence Equipment. Defence Standard 00-55, August 1995. http://www.dstan.mod.uk/ or http://www.seasys.demon.co.uk/.

[9] U.K. Ministry of Defence. Safety Management Requirements for Defence Systems Containing Programmable Electronics. Second Draft Defence Standard 00-56, August 1996. http://www.dstan.mod.uk/ or http://www.seasys.demon.co.uk/.

[10] F.A.C. Pinheiro and J.G. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, pages 52–64, 1996.

[11] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1:24–36, 1975.

[12] K.J. Ross. Models for configuration management of refinement calculus developments. In *Proceedings 7th Refinement Workshop*, pages 1–26. BCS-FACS, July 1996. also appears as SVRC TR 95-8.

[13] 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. Also appears as SVRC TR 93-3.

[14] K.J. Ross and P.A. Lindsay. Applying software configuration management techniques to formal development. Technical Report 95-12, Software Verification Research Centre, The University of Queensland, 1995.

[15] 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.

[16] W. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15, 1985.