SOFTWARE VERIFICATION RESEARCH CENTRE
SCHOOL OF INFORMATION TECHNOLOGY
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT
No. 97-13

Version and configuration management
of formal theories

Peter A. Lindsay  Owen Traynor

February 1997

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 via http://svrc.it.uq.edu.au
Version and configuration management of formal theories

Peter A. Lindsay       Owen Traynor

Abstract

This paper reports the results of an investigation of the problems of version management of systems of objects which themselves are under version control, and where there are complex consistency and completeness requirements. The issues are illustrated on a case study concerning management of a formal theory which consists of theorems and their proofs. Conclusions are drawn about a framework for fine-grained configuration management for formal methods of software development.

1 Introduction

1.1 Software Configuration Management (SCM)

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. In its most general form, Software Configuration Management concerns the control of all development artifacts throughout the system life-cycle, to preserve the definition of versions of components and the relationships among them [8].

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 [5, 13].

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 [20] or SCCS [16] 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.

1.2 SCM for Formal Methods

*Formal Methods* of software development have particular needs in relation to SCM. 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 [18].

One of the main practical problems currently hindering the uptake of Formal Methods, however, is the sheer number and complexity of its artifacts. For example, proofs are typically orders of magnitude larger than the programs they prove. By separating management of dependencies between formal entities from their construction, fine-grained SCM has much to offer Formal Methods, especially in the area of management of change. Most *Formal Development Support Environments* (FDSEs) currently do not provide specialised support for configuration management.

Direct adoption of standard approaches to SCM are not sufficient, however. Software configuration and version management for traditional software engineering focus on the requirements associated with the management of things like source code, object code, etc. In the context of formal development, the artifacts are more complex and more numerous and there is wider variety of dependencies between these artifacts. The concept of an automated “build” management system (upon which these traditional systems rely) is inappropriate for a FDSE, since the activities that bring configurations into consistent states cannot be fully automated.

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 [17, 18]. 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.

Early versions of the framework have concentrated on traceability and version management [12] for a number of reasons. First, we noted that it is often desirable to be able to store multiple versions of development artifacts – for example, in order to support team working, where different components of a system are developed largely independently and then brought back together [15]. Another reason is because of regulatory requirements: for example, one of the companies with which we have collaborated in the past is required to be able to reconstruct all development artifacts (including system software, analysis and test results) associated with the release of every one of its implantable medical devices.

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. For example, 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 [13]. Another example is the ability to trace (and reconstruct) artifacts which have been reused in different developments, such as components from standard libraries.

Other projects are looking at version management of structured objects (e.g. Goodstep [19], Merlin [14], UQ* [10, 21]) and have proposed generic approaches, with a good deal of success. However, the configuration consistency and completeness conditions they consider are not sophisticated enough to meet the above requirements.

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 FDSFs 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.

The approach taken here is 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 FDSEs, 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.

It is worth noting at this point that we consider version management as an integral part of the configuration management activity. Whereas it is common to have some form of configuration management without a version management system, version management systems rely critically on a configuration management framework for defining the collections of versions which denote a specific configuration.

1.4 The problem and its motivation

The approach will be illustrated here on version management of a formal theory, consisting of theorems and their proofs. This is an area of particular concern to FDSEs such as the B-Tool [1], Cogito [3] and mral [9], in which formal specifications have corresponding formal theories in which consequences of the specification are deduced. Version management is of critical importance in such systems because specifications – and hence the associated theorems – often change, yet the soundness of deductions depends on the specification and associated theory being in step with one another.

For some time, version control (at the theory level) has been used in theorem proving; when a change is made to a given theory, the old version is stored and the system is rebuilt with the new version. At the SVRC, we use CVS [4] for this task in the context of the Ergo theorem prover used with the Cogito system.

Similarly, most interactive theorem proving systems have some notion of configuration management at fine levels of granularity. For example, the Ergo theorem prover will warn the user if a circularity has been introduced during proof construction. This form of dependency management is a good illustration of the (simple) use of fine-grained configuration management principles. However, we are not aware of any theorem-proving systems which use notions of version control at this level of granularity.

The motivation for considering fine-grained version control of formal theories has come from a number of sources. Our experience in deploying complex theory structures for use in actual formal development, and our use of theorem provers (without fine-grained version control) to support the formal development activity, have illustrated the need for finer levels of configuration and version
control within the proof activity. In an industrial pilot project of Cogito [7] for example, the following scenario occurred a number of times.

During validation activities, from time to time the application developers would uncover a deficiency in a core theory. The maintainers of the theorem provers would then release a theory update which corrected the deficiency. Typically the deficiency would be in a small portion of a single theory, and an update would only be required for the current proof in that theory, since no other proofs had used that part of the theory so far. Since configuration control (and by implication version control) was at the theory level, however, no advantage could be taken of the fact that the theorem or definition being updated was used only in a single proof. Instead, the complete set of proofs for the development, and all consistency proofs for the core theories, had to be redone!

The same problem arose again at the application level. When a portion of a specification (say a definition) was modified, then all other specifications that relied on the changed specification (and by implication all the associated theories and proofs) required updating. Even for the relatively simple specification used in the particular development, this often meant days of extra work – work which would not be necessary if effective configuration management support had been in place.

1.5 This paper

This paper reports preliminary conclusions from a case study in adding fine-grained versioning to a simple theory store. As it happens, the case study is a good one for illustrating our general approach, because it is relatively concise, yet has complex consistency and completeness requirements. A key focus is the impact of the versioning policy on the consistency requirements of the theories, theorems and proofs that are the subject of version control. In conducting the analysis, we found that a variety of versioning policies were possible, all with different implications for the consistency of the theory store.

For simplicity, we have presented a simple model of theories and a very straightforward version control policy. However, even such a simple model has rather surprising implications for both the theory developer and user, which are discussed in more detail later.

Section 2 presents a basic model for version management of individual objects, covering notions of derivation history and freezing of version contents. The paper uses a variant of Object-Z [6] as the modelling language, for precision and conciseness. Section 3 defines the requirements for the case study. Section 4 models versions of theorems in terms of versions of statements and proofs. Section 5 models theories as collections of theorems. Section 6 defines the consistency and completeness conditions for the case study. Finally, the different models are brought together in Section 7, which treats version management of theories.
2 Version management

The model of version management considered here uses version trees to trace the development history of objects: see Fig. 1. Branching corresponds to derivation of variants of the parent version. (It is possible to add version merging to the model, but we have not done so here for simplicity.) The content of a version can be frozen, meaning the content cannot subsequently be changed. In particular, a version must be frozen before (or when) a new version is derived from it, since to allow subsequent changes would violate the traceability requirement.

![Version Tree Diagram](image)

Figure 1: An example version tree.

The rest of this section defines a generic class VersionTree[Content] of version trees, where Content is a generic set representing the kind of objects that are being put under version control.

2.1 Object state

The attributes of version trees are defined as follows:

- Each node in the 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 set \( \text{frozen} \) contains the labels of the frozen nodes.

• The mapping \( \text{parent} \) returns the (unique) parent of any given non-root node in the tree.

• One node is nominated as the current version.

\[
\begin{align*}
\text{root} &: \ VLabel \\
\text{deref} &: \ VLabel \rightarrow \ Content \\
\text{frozen} &: \mathbb{P} \ VLabel \\
\text{parent} &: \ VLabel \rightarrow \ VLabel \\
\text{current} &: \ VLabel \\
\text{nodes} &: \mathbb{P} \ VLabel \\
\text{currentcontent} &: \ Content \\
\text{nodes} &= \ \text{dom} \ \text{deref} \\
\text{currentcontent} &= \ \text{deref}(\text{current}) \\
\text{frozen} \cup \{\text{root}, \ \text{current}\} &\subseteq \text{nodes} \\
\text{dom} \ \text{parent} &= \ \text{nodes} \setminus \{\text{root}\} \\
\text{ran} \ \text{parent} &\subseteq \ \text{frozen}
\end{align*}
\]

Note that \( \text{nodes} \) and \( \text{currentcontent} \) are derived attributes: i.e., their values can be calculated from the other attributes. Such attributes do not need to be mentioned explicitly in operations’ delta sets.

The following predicate defines the initial values of version trees. The predicate takes the tree’s initial content \( \text{content?} \) as a parameter.

\[
\begin{align*}
\text{INIT} \\
\text{content?} &: \ Content \\
\text{deref} &= \{\text{root} \mapsto \text{content?}\} \\
\text{frozen} &= \emptyset
\end{align*}
\]

Note that the values of all the other attributes can be deduced from the above (e.g. \( \text{current} = \text{root} \)).

2.2 Operations

This section defines the basic operations that can be performed on a version tree. Note that each operation preserves the state invariant
- **FreezeNode**
  \[\Delta(frozen)\]
  \[v? : VLabel\]
  \[v? \in \text{nodes}\]
  \[\text{frozen}' = \text{frozen} \cup \{v?\}\]
  Freeze a given node.

- **DeriveNewVersion**
  \[\Delta(\text{nodes, parent, deref, frozen, current, newcontent?} : \text{Content})\]
  \[\text{FreezeNode[current/v?]}\]
  \[\exists \text{new : VLabel} \bullet\]
  \[\text{new} \notin \text{nodes}\]
  \[\text{parent}' = \text{parent} \cup \{\text{new} \mapsto \text{current}\}\]
  \[\text{deref}' = \text{deref} \cup \{\text{new} \mapsto \text{newcontent}\}\]
  \[\text{frozen}' = \text{frozen} \cup \{\text{current}\}\]
  \[\text{current}' = \text{new}\]
  Freeze the current node and create a new node with this as its parent and with the given content, making it the new current node.

- **DeriveNewCopy**
  \[\Delta(\text{nodes, parent, deref, frozen, current, newcontent?})\]
  \[\text{DeriveNewVersion}[\text{currentcontent/newcontent?}]\]
  Similar to DeriveNewVersion but takes a copy of the current content.

- **ChangeCurrent**
  \[\Delta(\text{current})\]
  \[v? : VLabel\]
  \[v? \in \text{nodes}\]
  \[\text{current}' = v?\]
  Change which node is considered the current node.

- **EditCurrent**
  \[\Delta(\text{deref})\]
  \[\text{content?} : \text{Content}\]
  \[\text{current} \notin \text{frozen}\]
  \[\text{deref}' = \text{deref} \oplus \{\text{current} \mapsto \text{content}\}\]
  Change the content of the given node, provided it is not frozen.
3 The case study: requirements

The case study is based on a (hypothetical) theory manager, in which a formal theory, consisting of theorems and their proofs, is put under version control. This section defines the basic requirements for the case study.

3.1 The objects and relationships to be managed

Theorems have statements and proofs. (In this simplified model, axioms are theorems with 'axiomatic' proofs.) We shall assume that proofs and statements can be edited separately, and so may get out of step (i.e., may not necessarily agree with one another). The internal structure of statements and proofs will not be considered here, other than to say that a proof may make reference to ('use') other theorems. This leads to a transitive dependency relation between theorems: see Fig. 2.

The system we consider is a flexible yet powerful one in which theorems can be proposed and used before their proofs are complete: Ergo and mural [9] are examples of such systems. For flexibility, we even allow the possibility that dependencies between theorems may be circular, although this is obviously a situation users would prefer to avoid (cf. Fig. 2). Making the model sufficiently general to support circularities means that a greater range of interaction models is possible, and more flexible user interfaces; however, these issues will not be addressed here.

To define the configuration consistency and completeness requirements we first make some definitions. A theory is a set of theorems. A theory is closed if all proofs use only theorems from within the theory (i.e., there are no dangling references). A theory is circular if there are loops in the dependency graph. A theory is complete if it is closed, non-circular, and all its theorems have complete and valid proofs.

We shall assume tools are available for:

- checking whether a proof is complete;
- extracting the theorems used in a proof; and
- editing statements and proofs.

Building on these, the paper specifies requirements for establishing and maintaining configuration consistency and completeness conditions. (Note however that logical consistency of the theory is not being considered here, just structural consistency.)

3.2 Versioning requirements

We want to be able to store multiple versions of statements and proofs of a given theorem, and store versions of proofs separately from statements. For
Example theorem statement
and proof

<table>
<thead>
<tr>
<th>Theorem A: $1 + 1 = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Proof:</strong></td>
</tr>
<tr>
<td>1. $1+1+1=3$</td>
</tr>
<tr>
<td>2. $1=0$</td>
</tr>
<tr>
<td>3. $1+1+0=3$</td>
</tr>
<tr>
<td>4. $1+1+0=1+1$</td>
</tr>
<tr>
<td>5. $1+1=3$</td>
</tr>
</tbody>
</table>

Example theorem dependency graph

Figure 2: (a) Example theorem, showing its statement and proof. (b) Example theory, showing dependency relation between theorems.
the purposes of the investigation we shall assume that there may be more than one proof corresponding to each statement, but that each proof has a unique corresponding statement.

A theorem instance is a particular version of a statement and a particular version of a proof of a theorem. Thus, what we said above was not quite accurate: proofs use theorem instances rather than simply theorems, and a theory is a set of theorem instances. The other definitions carry through mutatis mutandis.

To use the model of version management given in Section 2, we first need to come up with a versioning policy which includes an interpretation of when things can be frozen and what implications can be drawn (if any) from the fact that something is frozen. For the purposes of the investigation we adopt the following versioning policy:

- When a theorem instance is frozen, all of the theorem instances on which it depends should also be frozen. (Intuitively, the full justification of a theorem consists not only of its statement and proof, but also of the statement and proof of all the theorems on which it depends.)

- A version of a theory can be frozen only if it is complete.

This policy attempts to preserve the “semantic integrity” of theorems and theories. Of course, other versioning policies are possible for theory management, and this policy may be too stringent in practice — the point is that versioning policies are application-dependent.

### 3.3 Formal modelling

The following definitions are needed for the models in later sections of this paper. Let \( Stmt \) be the set of all possible statements of theorems, and let \( Proof \) be the set of all (possibly incomplete) proofs of theorems. \( Thmlist \) is the set of theorem instances (to be defined further below).

The basic tool-established relationships will be modelled as follows:

- the function \( uses : Proof \rightarrow \mathbb{P}Thmlist \) extracts the theorem instances which are used in a given proof;

- the relation \( proves : Proof \rightarrow Stmt \) checks whether a given proof is a complete and valid proof of a given statement.

The *mural* system [9] has such tools, for example.
4 The ‘theorem version history’ object class

This section defines a class ThmVHistory for the version histories of particular theorems.

4.1 Object state

![Diagram of version trees for a theorem]

Figure 3: Example version trees for a theorem

A theorem version history is a tree of versions of statements and proofs of a particular theorem, together with an indication of the correspondence between proofs and statements (called ‘justifies’ here): see Fig. 3. Note that the ‘justifies’ relation simply notes which statement corresponds to which proof in the configuration: it is a structural relation only, and nothing can be inferred from it about whether the given proof actually ‘proves’ the given statement.

\[
\begin{align*}
stmts &: \text{VersionTree}[\text{Stmt}] \\
proofs &: \text{VersionTree}[\text{Proof}] \\
justif &: \text{VLabel} \rightarrow \text{VLabel}
\end{align*}
\]

\[
\forall p : \text{proofs}\text{.nodes} \bullet \exists_1 s : \text{stmts}\text{.nodes} \bullet \text{justif}(p, s)
\]

\[
\text{dom justif} = \text{proofs}\text{.nodes}
\]

\[
\text{ran justif} = \text{stmts}\text{.nodes}
\]

In accordance with the requirements defined in Section 3, the invariant says that ‘justifies’ is a many-one correspondence: i.e., each proof has exactly one corresponding statement, and each statement has at least one corresponding proof, with no dangling references.

A new theorem version history is created by supplying an initial statement and proof:
4.2 Operations

- **FreezeThmVersion**
  \[ \Delta(stmts, proofs) \]
  \[ s?, p? : VLabel \]
  \[ \text{justif}(p?, s?) \]
  \[ stmts.FreezeNode[s?/v?] \]
  \[ proofs.FreezeNode[p?/v?] \]
  Freeze the given statement and proof of the theorem, provided they correspond.

- **EditStmt**
  \[ \Delta(stmts) \]
  \[ new? : Stmt \]
  \[ stmts.EditCurrent[new?/content?] \]
  Edit the current statement, provided it is not frozen. (The precondition \( stmts.current \notin stmts.frozen \) is carried over from \( EditCurrent. \))

- **EditProof**
  \[ \Delta(proofs) \]
  \[ new? : Proof \]
  \[ proofs.EditCurrent[new?/content?] \]
  Edit the current proof, provided it is not frozen.

- **ChangeCurrentThmVersion**
  \[ \Delta(stmts, proofs) \]
  \[ s?, p? : VLabel \]
  \[ \text{justif}(p?, s?) \]
  \[ stmts.ChangeCurrent[s?/v?] \]
  \[ proofs.ChangeCurrent[p?/v?] \]
  Change which statement and proof are considered the current versions for the theorem, provided they correspond.

5 The ‘theory’ object class

This section defines a class Theory for collections of theorem instances.
5.1 Preliminaries

A theorem instance consists of the name and statement of a theorem and two version labels, for selecting the particular statement and proof from the theorem’s version trees, respectively.

\[
\begin{array}{l}
\_ThmInst \\
name : ThmName \\
sversion : VLabel \\
pversion : VLabel \\
\end{array}
\]

5.2 Object state

A theory is a set of theorem instances such that no theorem name appears more than once (i.e., a theory contains at most one version of each theorem). For convenience, a derived attribute \textit{workingset} is introduced to stand for the ‘working set’ of (names of) theorems in the theory.

\[
\begin{array}{l}
\_thms : F ThmInst \\
\_workingset : F ThmName \\
\forall a, b : thms \bullet a.a.name = b.a.name \Rightarrow a = b \\
\end{array}
\]

Initially, a theory is empty:

\[
\begin{array}{l}
\_INIT \\
\_thms = \emptyset \\
\end{array}
\]

5.3 Operations

\[
\begin{array}{l}
\_AddThm \\
\Delta(thms) \\
a? : ThmInst \\
\forall a'.name \in workingset \Rightarrow \\
\text{thms}' = (\text{thms} \setminus \{a\}) \cup \{a'\} \\
\text{where } a = (\mu a : thms \mid a.a.name = a?.name) \\
a?.name \notin workingset \Rightarrow \\
\text{thms}' = \text{thms} \cup \{a?\} \\
\end{array}
\]

Add the given theorem instance to the theorem, replacing the instance of the same name if necessary.
6 Theorem networks

This section defines the configuration consistency and completeness properties for the case study. First we define a theorem map to be a collection of theorem names and their corresponding version histories:

\[ \text{ThmMap} \equiv \text{ThmName} \rightarrow \text{ThmVHistory} \]

Theorem maps contain the contextual information required for the definitions below.

6.1 Auxiliary functions

The following relation represents the theorem instances which belong to (are valid in) a given theorem map:

\[ \text{isValidInst} : \text{ThmInst} \rightarrow \text{ThmMap} \]

\[
\text{isValidInst}(a, \text{lookup}) \iff \\
\begin{align*}
\text{a.name} &\in \text{dom lookup} \\
\text{a.sversion} &\in \text{t.stmts.nodestrings} \\
\text{a.pversion} &\in \text{t.proofs.nodestrings} \\
\text{t.justify(a.pversion, a.sversion)} \\
\text{where } t &= \text{lookup}(a.nname)
\end{align*}
\]

(Where variable declarations in Z definitions are obvious, they are dropped for brevity.) The following function returns the proof corresponding to a (valid) theorem instance:

\[ \text{getProof} : \text{ThmInst} \times \text{ThmMap} \rightarrow \text{Proof} \]

\[
\begin{align*}
\text{dom getProof} &= \text{isValidInst} \\
\text{getProof}(a, \text{lookup}) &= \text{t.proofs.deref}(a.pversion) \\
\text{where } t &= \text{lookup}(a.nname)
\end{align*}
\]

The function \( \text{getStmt} : \text{ThmInst} \times \text{ThmMap} \rightarrow \text{Stmt} \) is defined similarly.

6.2 Dependency checking functions

We say a theorem instance \( a \) depends directly on theorem instance \( b \) if the proof of \( a \) uses \( b \):
\[ \text{ddo} : \mathbb{P}((\text{ThmInst} \times \text{ThmInst} \times \text{ThmMap}) \times (\text{ThmInst} \times \text{ThmMap}) \times \text{ThmMap}) \]

\[ \text{ddo}(a, b, \text{lookup}) \iff \text{isValidInst}(a, \text{lookup}) \land b \in \text{uses}(\text{getProof}(a, \text{lookup})) \]

A dependency chain is a sequence of theorem instances, each of which depends directly on the next one in the chain:

\[ \text{isDepChain} : (\text{seq ThmInst}) \rightarrow \text{ThmMap} \]

\[ \text{isDepChain}(a, \text{lookup}) \iff \forall i : 1 \ldots \#a - 1 \bullet \text{ddo}(a(i), a(i + 1), \text{lookup}) \]

Theorem instance \( a \) depends on theorem instance \( b \) if there is a non-trivial dependency chain from \( a \) to \( b \):

\[ \text{dependsOn} : \mathbb{P}((\text{ThmInst} \times \text{ThmInst} \times \text{ThmMap}) \rightarrow (\text{ThmInst} \times \text{ThmInst} \times \text{ThmMap}) \times (\text{ThmInst} \times \text{ThmMap}) \times \text{ThmMap}) \]

\[ \text{dependsOn}(a, b, \text{lookup}) \iff \exists a : \text{seq ThmInst} \bullet \#a \geq 2 \land \text{isDepChain}(a, \text{lookup}) \land \text{head as} = a \land \text{last as} = b \]

The supports of theorem instance \( a \) is the set of all theorem instances on which \( a \) depends

\[ \text{supporters} : \text{ThmInst} \times \text{ThmMap} \rightarrow \mathbb{P} \text{ThmInst} \]

\[ \text{supporters}(a, \text{lookup}) = \{ b : \text{ThmInst} \mid \text{dependsOn}(a, b, \text{lookup}) \} \]

A theory is closed if it has no dangling references (i.e., it contains all supporters of its theorem instances):

\[ \text{isClosed} : \text{Theory} \rightarrow \text{ThmMap} \]

\[ \text{isClosed}(T, \text{lookup}) \iff \forall a : T.\text{thms} \bullet \text{supporters}(a, \text{lookup}) \subseteq T.\text{thms} \]

A theory is circular if it contains a theorem instance which depends on itself:

\[ \text{isCircular} : \text{Theory} \rightarrow \text{ThmMap} \]

\[ \text{isCircular}(T, \text{lookup}) \iff \exists a : T.\text{thms} \bullet \text{dependsOn}(a, a, \text{lookup}) \]

### 6.3 Theorem checking functions

The following checks whether a particular theorem instance is proven:

\[ \text{isProven} : \text{ThmInst} \rightarrow \text{ThmMap} \]

\[ \text{isProven}(a, \text{lookup}) \iff \text{isValidInst}(a, \text{lookup}) \land \text{proves}(\text{getProof}(a, \text{lookup}), \text{getStmt}(a, \text{lookup})) \]

Note that, in this model, being ‘proven’ just means that the immediate proof is complete. Since the proof may however depend on proofs which are not
themselves complete, being proven in this sense is not sufficient to fully justify a theorem.

A theorem is *established* if it is proven and everything on which it depends is proven:

\[
\text{isEstablished} : \text{ThmInst} \rightarrow \text{ThmMap}
\]

\[
\text{isEstablished}(a, \text{lookup}) \iff
\text{isProven}(a, \text{lookup}) \land \forall b : \text{supporters}(a, \text{lookup}) \bullet \text{isProven}(b, \text{lookup})
\]

Again, a theorem may be ‘established’ and yet not be valid: e.g., if reasoning is circular.

A theory is *complete* if it is closed and non-circular, and all its theorems are established:

\[
\text{isComplete} : \text{Theory} \rightarrow \text{ThmMap}
\]

\[
\text{isComplete}(T, \text{lookup}) \iff
\text{isClosed}(T, \text{lookup}) \land \neg \text{isCircular}(T, \text{lookup})
\land \forall a : T.thms \bullet \text{isEstablished}(a, \text{lookup})
\]

7 The ‘theory configuration’ object class

This section describes a class *TheoryConfig* for version management of a theory. This establishes two levels of version: versions of theorems within the theory, and versions of the theory itself.

7.1 Object state

A *theory configuration* (or configuration, for short) consists of a theorem network, together with versions of a theory.

\[
\begin{align*}
\text{lookup} &: \text{ThmMap} \\
\text{theory} &: \text{VersionTree}[\text{Theory}] \\
\text{currentTheory} &: \text{Theory}
\end{align*}
\]

\[
\text{currentTheory} = \text{theory.currentcontent}
\]

Note that the ‘working’ version of a theory and the ‘current’ version of a theory may get out of step: i.e., theorem instances in the current version are not necessarily the current versions of the theorems in the working set.

Initially the configuration is empty.
7.2 Operations

The following are some of the operations that would be available to users:

_AddNewThm_________
\[ \Delta(\text{lookup}) \]
\[
\begin{array}{rcl}
\text{name} : & \text{ThmName} & \\
\text{stmt} : & \text{Stmt} & \\
\text{proof} : & \text{Proof} & \\
\end{array}
\]
\[ \text{name} \notin \text{dom lookup} \]
\[ \text{lookup}' = \text{lookup} \uplus \{ \text{name} \mapsto t \} \]
\[ \text{where } t = \text{ThmVHistory.INIT} \]

_AddThmToTheory_________
\[ \Delta(\text{theory}) \]
\[ a? : \text{ThmInst} \]
\[ \text{isValidInst}(a?, \text{lookup}) \]
\[ \text{theory.current} \notin \text{theory.frozen} \]
\[ \text{currentTheory.AddThm} \]

_RemoveThm_________
\[ \Delta(\text{theory}) \]
\[ a? : \text{ThmInst} \]
\[ \text{isValidInst}(a?, \text{lookup}) \]
\[ \text{theory.current} \notin \text{theory.frozen} \]
\[ \text{currentTheory.RemoveThm} \]

_FreezeThm_________
\[ \Delta(\text{lookup}) \]
\[ a? : \text{ThmInst} \]
\[ \text{isValidInst}(a?, \text{lookup}) \]
\[ \text{and } \forall a : \text{supporters}(a?, \text{lookup}) \cup \{ a? \} \Rightarrow \text{lookup}(a.\text{name}).\text{FreezeThmVersion}[a.\text{sversion}, a.\text{pversion} / s?, p?] \]

_addNewThm = \emptyset
theory.INIT[T/\text{content}] \text{ where } T = \text{Theory.INIT}
\[ \begin{align*}
\text{\_FreezeTheory} & \quad \Delta(\text{lookup, theory}) \\
& \quad \text{isComplete}(T, \text{lookup}) \\
& \quad \land \ a : T.\text{thms} \bullet \text{FreezeThm}[a/a?] \\
\quad & \quad T.\text{FreezeCurrent} \\
& \quad \text{where } T = \text{currentTheory}
\end{align*} \]

\[ \begin{align*}
\text{\_DeriveNewTheory} & \quad \Delta(\text{theory}) \\
& \quad \text{theory.current } \in \text{theory.frozen} \\
& \quad \text{theory.DeriveNewCopy}
\end{align*} \]

Freeze all theorem instances in the current theory, provided it is complete.

Derive a new theory by taking a copy of the current theory; as a precondition we require the current theory to be frozen.

Other operations would include:

- changing what is considered the current version of a theory or theorem;
- editing (non-frozen) statements and proofs;
- deriving new versions of statements and proofs;
- changing the version of the theorem referenced in a proof.

### 7.3 Adherence to versioning policy

It is easy to show that the following properties are “behavioural invariants” of the class (i.e., they are true initially and are preserved by all enabled operations):

\[ \begin{align*}
\forall t : \text{ran lookup} & \bullet \text{justif}(t.\text{proofs.current}, t.\text{stmts.current}) \\
\forall t : \text{ran lookup} & \bullet \forall p : \text{ran}(t.\text{proofs.deref}) \bullet \forall a : \text{uses}(p) \bullet \text{isValidInst}(a, \text{lookup}) \\
\forall T : \text{ran(\text{theory.deref})} & \bullet \forall a : T.\text{thms} \bullet \text{isValidInst}(a, \text{lookup}) \\
\forall i : \text{\_theory.frozen} \bullet \text{let } T = \text{deref}(i) \text{ in} & \\
& \quad \text{isComplete}(T, \text{lookup}) \land \forall a : T.\text{thms} \bullet \text{let } t = \text{lookup}(a.\text{name}) \text{ in} \\
& \quad a.\text{version } \in t.\text{stmts.frozen} \land a.\text{version } \in t.\text{proofs.frozen}
\end{align*} \]

In words, the behavioural invariant says:

1. the current statement and proof of any theorem in the configuration are always in correspondence;

2. all proofs in the configuration use only theorem instances which are valid in the configuration (i.e., there are no dangling references in proofs);

15
3. all theorem instances in all theory versions are valid in the configuration (i.e., no dangling references in theories); and
4. all frozen versions of the theory are complete and all of their theorems are frozen.

Properties 1–3 make explicit the configuration consistency conditions implicit in Section 3. Property 4 shows that our model satisfies the versioning policy defined in Section 3.

8 Conclusions

8.1 Summary

This paper described a case study in fine-grained version and configuration management. The subject of the case study was the management of a formal theory consisting of theorems and proofs. 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 a theory as first-class citizens in the context of configurations. This allows substantial flexibility in the way in which consistency of an overall theory store is determined, as well as focusing attention on the specific objects undergoing change. The second benefit comes from the actual versioning of the theory 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.

A key issue arising in the case study concerned requirements associated with the definition of a versioning policy. We presented a generic versioning class which was used as the basis for versioning of all objects within configurations. As we have shown, the definition of the basic versioning framework is insufficient in itself to guarantee the consistency of configurations. Thus the versioning policy must be constructed with some care, to ensure that consistency criteria of the overall system are met. In our example, this required that careful consideration be given to the operations that implement the versioning policy model.

We were required to define additional constraints in the pre-conditions of the versioning operations that used primitive versioning mechanisms, to ensure that these operations preserved the configuration consistency criteria.

The simple version control model illustrated in this paper is adequate for simple versioning policies (and does solve the problems we have encountered and outlined in section 1.2). However, due to the consistency constraints present in
the model, it has deficiencies which would make the versioning model outlined here impractical for large development environments:

- Some principles of abstraction and independence are violated. For example, freezing a theorem results in all supporters being frozen, even those whose proofs are still under construction. When the latter are subsequently modified (by deriving and editing new versions), it will be necessary to re-link all proofs which refer to the old version. We have identified a solution to this problem, however it does complicate the overall model.

- The versioning offered in our model provides an opportunity to identify the minimal amount of rework required to accommodate a change in a theory. However, this does require strategic decisions to be made, such as whether or not to update the version of a theorem referenced in some other proof when a new version of the theorem is created. This may in turn complicate the overall theory structure: for example, it may be desirable to allow different versions of theorems to be used simultaneously in a single version of a theory store. (Our model currently precludes this.) The desire for such flexibility has to be balanced against the complication it adds to the process of theory maintenance (e.g. when freezing a theory).

Using versioning in the way proposed in this paper offers developers the opportunity to make decisions as to how to react to a given change and to control the immediate extent to which a change (new version) will impact on the rest of the system. Without the versioning, such options would be extremely difficult (or impossible) to provide.

### 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). The case study considered a single theory. It's clear that, by supporting more sophisticated theory structuring mechanisms (such as theory interpretation and instantiation), 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 theory structures, to provide a basis for the flexible theory construction and maintenance facilities suitable for large-scale formal 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 theory store 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 (but without versioning) [11], and have extended them to the case study presented in this paper.

Finally, we are prototyping tools to support our fine-grained configuration framework using object-oriented database technology [2].

8.2.1 Acknowledgements

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

References


