META-DATA VERSIONING

Greg Adamski
L-3 Communications, Telemetry-West

ABSTRACT

Telemetry missions spanning multiple years of tests often require access to archived configuration data for replay and analysis purposes. The needs for versioning vary from simple file-naming conventions to advanced global database versioning based on the scale and complexity of the mission. This paper focuses on a flexible approach to allow access to current and past versions of multiple test article configurations. Specifically, this paper discusses the characteristics of a versioning system for user-friendly and feature-rich solutions. It analyzes the tradeoffs of various versioning options to meet the needs of a given mission and provides a simple framework for users to identify their versioning requirements and implementation.

KEY WORDS

Meta-Data, Versioning, SQL, Revision Control

INTRODUCTION

Telemetry missions may span multiple years and consist of many individual tests. The configuration of the system will most likely change over time. Additionally, users may want to assign arbitrary meta-information to each test to facilitate future data retrieval and cross test data analysis. The management of such a system presents a very real issue: all historical versions of system configuration must be stored, but more importantly, there has to be a mechanism to easily recall those settings and apply them when needed. The role of the versioning subsystem discussed in this paper is exactly this. It allows user to store historical versions of their system configurations and let them retrieve when needed.

USE CASE OVERVIEW

To better understand user behavior, the following set of use cases has been created. This is not an exhaustive list of user interactions with a versioned system. However, the use cases presented below cover a majority of operational scenarios and were a basis for the system design and implementation.

1. The user wants to create a new project
    a. The user is starting from scratch

   b. The user wants to use existing data as a baseline
      i. The user has databases from a previous project he/she wants to re-use in this new project
         1. The user wants to re-use the full project as a starting point
         2. The user only wants to use specific data from the project
            a. A full module database
            b. Specific parameters only
            c. Module configuration only
            d. Display overlays only
      ii. The user wants to import external data
         1. The user wants to import databases from an external TMATS file
            a. The user wants to overwrite existing database information
            b. The user wants to merge external information with existing database information
         2. The user wants to import databases from an external non-TMATS source (Excel, SQL, text file, etc)
            a. The user wants to overwrite existing database information
            b. The user wants to merge external information with existing database information
2. The user wants to save his/her work and stop using the system
   a. The user just wants to save the current setup
   b. The user has a working setup and wants to take a “snapshot” (to recall it later if needed)
   c. The user doesn’t care and wants to discard the current setup
3. The user wants to open an existing project
   a. The user looks through the list of existing projects
   b. The user uses keywords to find a specific project
4. The user found a project and opens it
   a. The user wants to use the latest saved setup
   b. The user wants to open a previous snapshot and make changes
      i. The user knows which snapshot to open
      ii. The user wants to compare versions of previous snapshots
5. The user wants to backup/restore one or more projects
   a. The user wants to backup/restore all existing projects
   b. The user wants to backup/restore a specific project
6. The user wants to export a project to another computer
   a. The user wants to export measurements, configurations, display overlays, etc
   b. The user only wants to export a subset of a project setup
7. The user exports a project from a main system and imports the data on a remote system
   a. The user make changes to the project on the remote system
      i. Someone else makes changes to the project on the main system
      ii. No one changes the state of the project on the main system
   b. The user brings his/her project back and tries to integrate with all the possible changes
      i. The main system is not changed

        1. The user brings the project back and import the changes to the main system (no merge / integration necessary)
    ii. The main system has changed
        1. The user merges with or overwrites the data on the main system

SYSTEM ARCHITECTURE COMPARISON

Architectures of a telemetry project can be divided into three categories based on the coupling of individual components:

Shared modules model

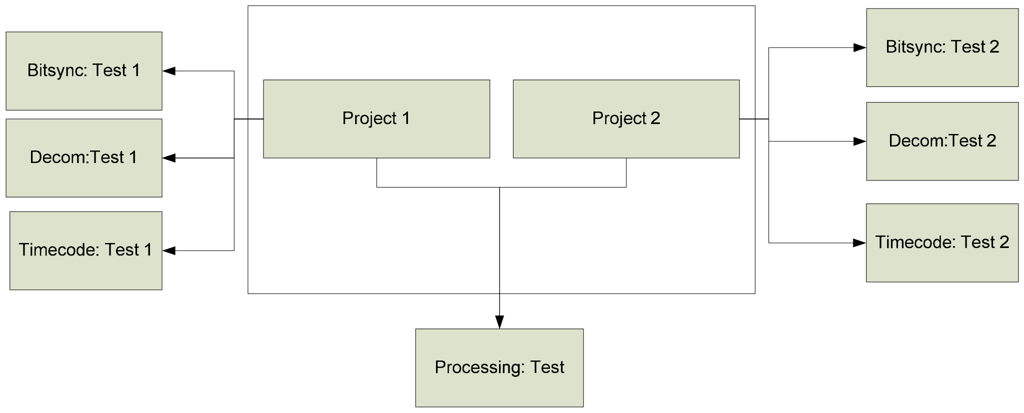


Figure 1. Shared modules

In the shared modules model, module databases exist independently of the project. The Project only keeps references to a module database definition thereby allowing other projects to share a module database.

Self-contained project with module selection model

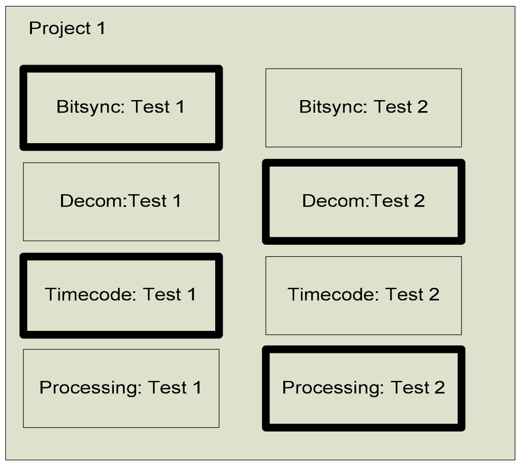


Figure 2. Self contained project with module selection; Bold line indicates active configuration

In a self contained project model, module databases belong to a particular project. This means that changes to other projects do not affect the integrity of the current project. This model allows users to have more than one configuration for each module that are individually selectable within the current project.

Self-contained project model

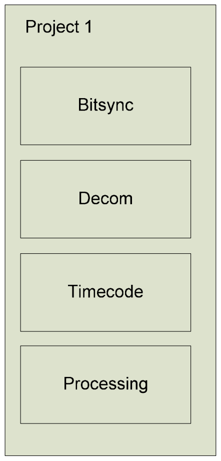


Figure 3. Fully self -contained project

In this model project may have only one configuration per module. This assures that the project information is consistent at all times, but limits user flexibility.

	Shared Module	**Self contained project with module selection**	**Self-contained project**
Flexibility	Once created modules can be user anywhere; changes to modules propagate automatically	Modules may not be shared across projects; user may still select different modules to test different scenarios	A new project has to be created to test any variations
System Integrity	Not maintained, left to the user.	Project-level. Changes to one project do not affect any other project	Full integrity maintained at all times
Usage Complexity	Sharing module databases may be confusing and lead to errors, requires rigid user process and methodology	No guarantee of system integrity due to multiplicity of databases may lead to configuration errors	Simple
Storage	No redundancy;	Creating new	Each change that

requirements	maximum data reuse	project means recreating all data; different scenarios may exist in one project	needs to be tracked requires copying of all the information

Table 1. Fully self -contained project

VERSIONING PROJECT VS. MODULE

A Project, as a collection of Modules may also change over time. Database assignments may change, and modules may be added or removed from a project. This means that the system must keep track of both project and module versions. How much of this functionality should be exposed to the user will be discussed later. The design has to consider the following problems:

a. Should the user be allowed to have modules with different configuration versions in a single project?
b. If a module database can be shared across different projects, how to keep track of version consistency?

VERSIONING SCHEMES

a. Record Level Versioning

When looking at the system configuration at any point in time, it is a sum of initial configuration plus subsequent changes. From system creation, only changes to objects were being recorded. The example in Table 1 shows how changes were recorded.

Step	Operation	State

Initial state	-	Object1 = 1500
		Object2 = 2000

Change 1	object1 = 1000	Object1 = 1000
		Object2 = 2000

Change 2	object1 = 1700	Object1 = 1700
		Object2 = 2000

Change 3	object2 = 3000	Object1 = 1700
		Object2 = 3000

Table 2. Record level versioning

This provides the ability to restore system configuration from any point in time, but causes problems when attempting to load a configuration, modify and then store it to the database. Since continuity of changes is not maintained, integrity of the system can not be guaranteed. On the other hand introducing a concept of branching turned out to be too difficult for users without experience in software development.

b. Table Level Versioning

An alternative approach to versioning is to store a complete snapshot of the system whenever a version is saved. This allows the user to maintain system integrity, but reduces flexibility and increases storage requirements of such a system. The term table-level versioning refers to the system's ability to snapshot complete data tables when a version is saved.

c. Hybrid Approach

In our system, we decided to use a hybrid approach that gives us the best of both worlds. Storage area is divided into two parts: work data and archival storage. Work data is stored using record-level versioning and archived versions use table- level versioning. Both areas are constructed in such a way, that the access to them is possible using the same API calls. The difference is that while the user has the ability to modify work data at will, access to stored versions is read only. Modifications to stored versions are possible only by copying them to the work area and modifying them there.

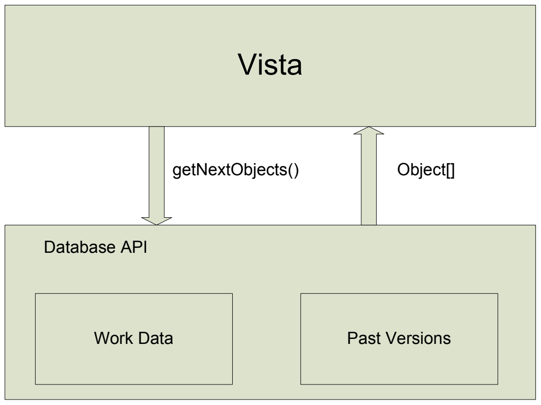


Figure 4. Fully self -contained project

STORAGE OVERVIEW

The system stores project information inside a relational database allowing us to provide shared access to the data from all locations on a local network. Database objects are not accessed directly though, but through an API that is responsible for mapping JAVA objects to database tables. Versioning is built on top of that API. Applications requesting an older version of an object/module/project database must use *setVersion()* call before requesting objects. After that, all objects returned from the database will come from the specified version.

Version Storage

Storing a version is only a part of the problem. Getting information back is another, especially if many versions of a single project have been created. While each object in the database is marked with a version number, versions themselves contain a significant amount of meta-information:

- Version Number
  By default versions are numbered using <major>.<minor> notation, but users may change version number at will
- Version Date
  Version date is typically an indicator of when the test took place. This information also allows us to load a correct system configuration when playing back an archival tape.
- Version Description
  Description is a free form field. Users can enter anything they might want to know/track about this particular version
- User-defined Tags
  Users have the ability to specify a number of enumerable fields that are specific to their test. Figure 5 presents sample user defined field combination. This is a powerful feature providing flexible retrieval capabilities (custom filtering, sorting, etc).

Providing well-defined meta information allows users to easily retrieve correct versions when needed. Figure 8 shows an example version picker dialog. The **Filter** panel allows users to quickly limit versions that are displayed to those that meet specific criteria be it version number, time span, or user defined tags.

Sharing databases

As mentioned before, adding version information to the project leads to complications in dependencies between databases shared across projects. To simplify this, the current design does not allow users to share archived versions across projects. Users are still free to use shared work data across project, which is not recommended though and will be disabled in future releases.

PERFORMANCE IMPLICATIONS

Versioning adds an additional layer of complexity on top of the existing database storage. Our research has shown that the design chosen has minimal impact on data retrieval performance. Figure 4 shows a linear increase in load time for a project with 10 objects. The slope of the curve is small enough that the performance penalty can be easily disregarded in typical scenarios.

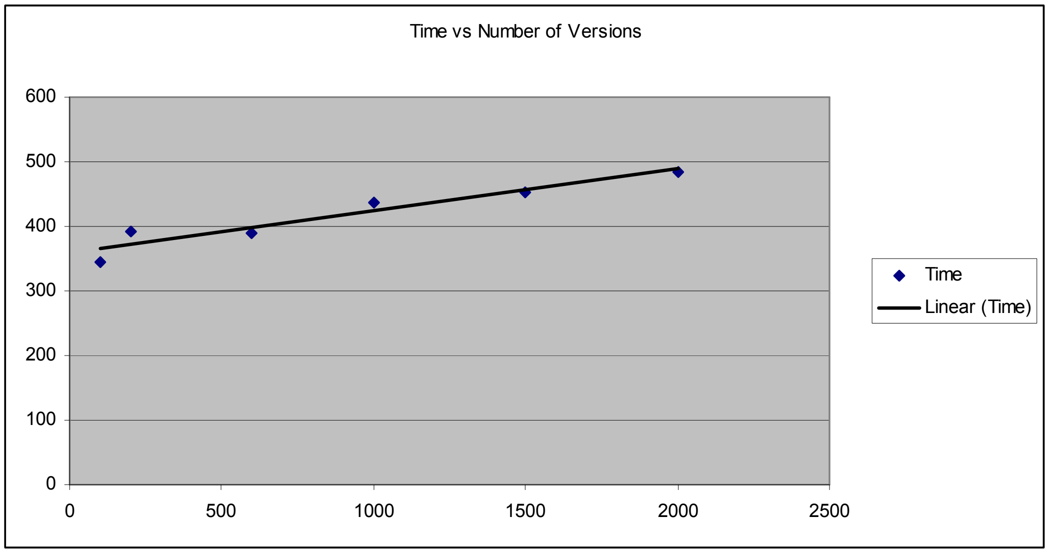


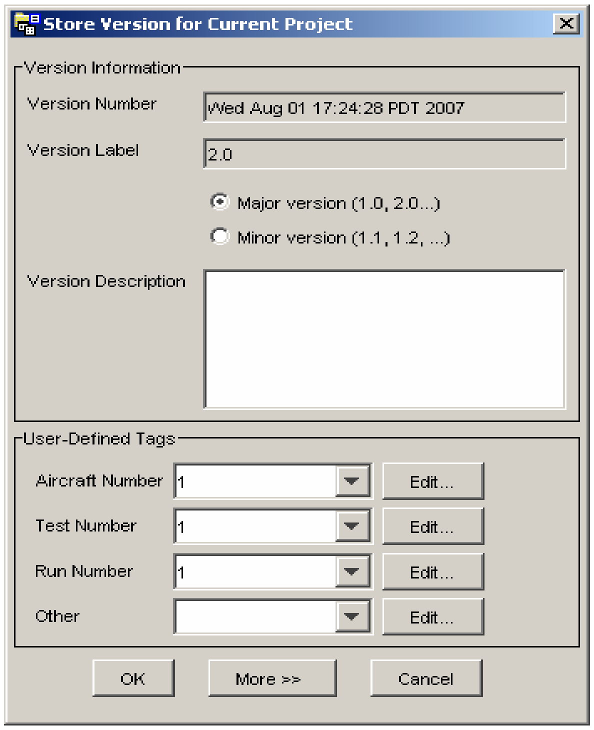
Figure 5. Changes of load time with increase of the number of versions

API CHANGES

An important factor when designing a versioning scheme was how big of an impact it would have on the rest of the system. How much would the system have to modified to support versioning and how much testing would have to be performed to make sure the system worked correctly. That is why the API was designed to be transparent to applications that do not want to take advantage of versioning. Only code that is responsible for transferring configuration data in or out of the system (e.g. XML Import/Export, TMATS Import/Export) needs to be aware of the internal workings of the versioning scheme. Most other applications use standard calls for retrieving data that existed before. Only in a situation when a specific version is needed, do they need to call setVersion().

GUI DESIGN

When designing GUIs for this task, contradicting requirements had to be taken into account. On one hand GUIs had to be simple, not to overcomplicate an already complex notion of version management. On the other, they had to give users enough flexibility to allow advanced manipulation of data on systems with many versions. A decision was made to hide most of the options to typical users, while at the same time allowing advanced users to access all the functionality they may need on demand. After dialogs open, users are presented with a limited set of settings. Advanced options only show after “More” is pressed. Figure 5, Figure 6, Figure 7 and Figure 8 demonstrate this approach. It is also important to note that users who do not want versioning, do not need to worry about it even if it is enabled.


Store Version for Current Project

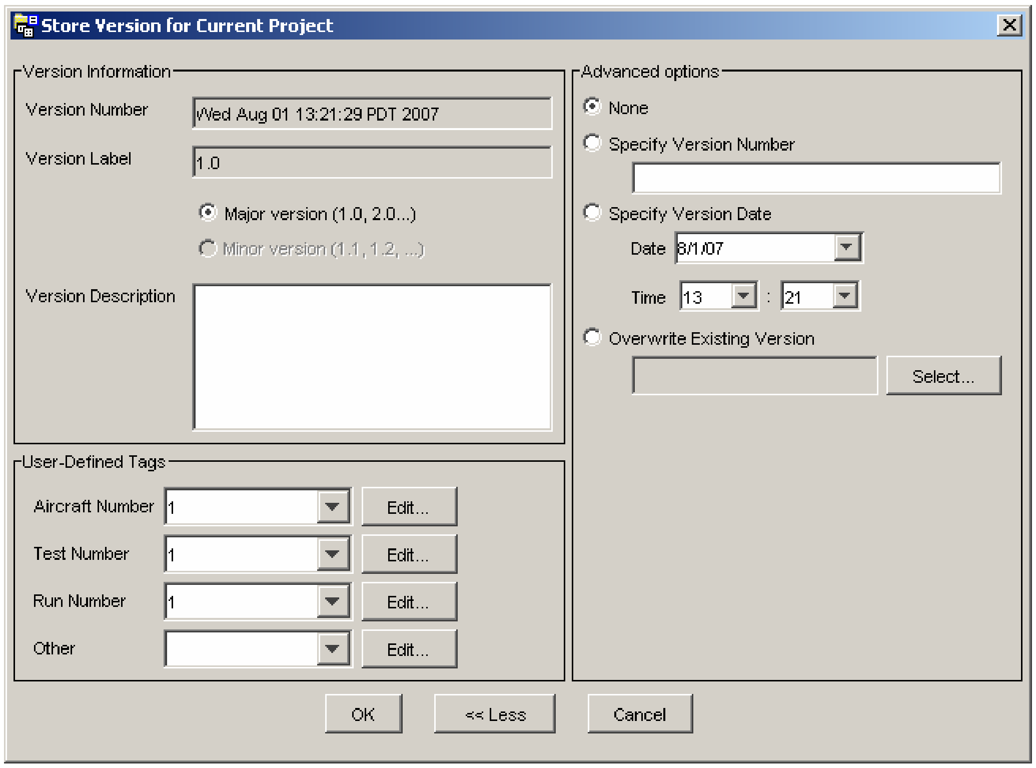
Version Information

Version Number	Wed Aug 01 17:24:28 PDT 2007
Version Label	2.0
	[✓ Major version (1.0, 2.0...)] Minor version (1.1, 1.2, ...)
Version Description	

User-Defined Tags

Aircraft Number	1	Edit...
Test Number	1	Edit...
Run Number	1	Edit...
Other		Edit...

Figure 6. Storing new version of a project – simple view


Store Version for Current Project

Version Information

Version Number	Wed Aug 01 13:21:29 PDT 2007
Version Label	1.0
	[✓ Major version (1.0, 2.0...)] Minor version (1.1, 1.2, ...)
Version Description	

User-Defined Tags

Aircraft Number	1	Edit...
Test Number	1	Edit...
Run Number	1	Edit...
Other		Edit...

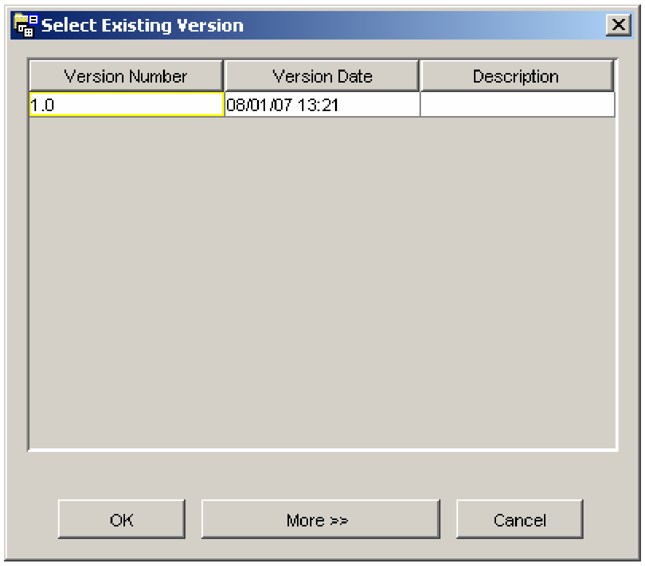
Advanced options

[✓ None] Specify Version Number Specify Version Date Overwrite Existing Version

Date 8/1/07

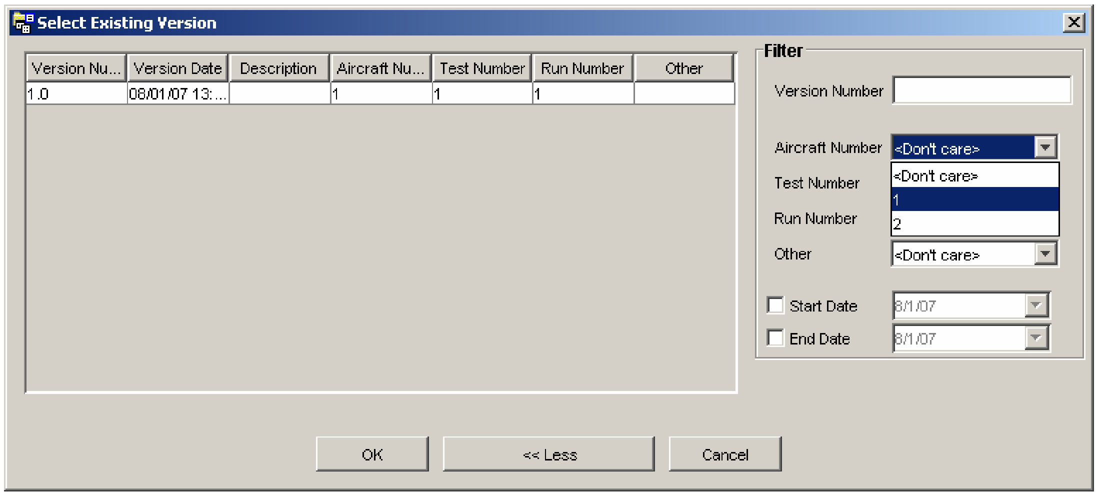
Time 13 : 21

Figure 7. Storing new version of a project – advanced view



Version Number	Version Date	Description
1.0	08/01/07 13:21	

Figure 8. Retrieving a stored version – simple view



Version Nu...	Version Date	Description	Aircraft Nu...	Test Number	Run Number	Other
1.0	08/01/07 13:...		1	1	1	

Version Number
Aircraft Number: <Don't care>
Test Number: <Don't care> / 1 / 2
Run Number
Other: <Don't care>
Start Date: 8/1/07
End Date: 8/1/07

Figure 9. Retrieving a stored version – advanced view

CONCLUSION

Meta-Data versioning is not a trivial task. It is crucial to identify the true usage scenarios that must be addressed and make compromises along the way to promote ease-of-use of the design. A hybrid approach seems to provide the most effective solution for typical telemetry system versioning needs.