Mature Model Management

Steven Kelly, MetaCase

With the boom in modeling, everybody is trying to manage models with existing versioning tools and practices. And everybody is suffering: modelers, VCS vendors and modeling tool vendors alike. In this article we take a step back and look at what is actually needed for model management, and how a novel approach, designed from scratch for models rather than text, has fared in practice. We will examine what old problems of model, variant and version management it solves, and how. On the counter side we look at what new problems Domain-Specific Modeling Languages raise for model management, and how to address them. These results are based on and will be illustrated by examples from a variety of industries, including telecom, medical and finance. From these and other experiences over the last fifteen years, we will also see which ideas sound great in theory but don't work out in practice — and why.

1 Versioning text and models

From humble beginnings as simple text file archivers, Version Control Systems have grown to play an integral role in software development, covering all manner of management of source artifacts. Modern VCS functionality handles three main tasks [Alt09]:

- **Archive**: maintaining older versions and allowing developers to return to them
- **Collaboration**: integrating the work of multiple developers
- **Branching**: handling long term parallel variants and configuration management.

This set of tasks, and the ability of VCSs to handle them effectively, has grown over the decades; indeed, major new VCSs appear and are adopted about once a decade. The original SCCS (1972) lacked collaboration and effective branching, and was superceded by RCS (1982) which added those areas. Its successors CVS (1990) and Subversion (2000) represent continuous, incremental improvement rather than radical changes. Distributed VCSs, such as Mercurial and Git, have proved popular in open source development, but have not been widely adopted in the more centralized environment of in-house commercial development, which we shall focus on here.

1.1 Text vs. objects

All of these VCSs have in common a focus on textual files; other kinds of files may be stored, but major functions will only work for text. They will thus work fine with textual source code like C or Java, although they have no understanding of the semantics: a change to the text “Employee” inside a string is treated exactly the same as a change to a class name “Employee” in a method signature. The latter change would normally be accompanied by a similar change in every other place where the Employee class is used, and also to the name of the Employee class file. Such wholesale changes can be made in one operation with refactoring tools, but from the point of view of the VCS they appear as multiple unrelated edits across many files.

The root of the problem lies not with the VCS, but with the use of text as a data structure to store programs. Text is a simple one dimensional array of characters, whereas a program is in fact a complex graph of nodes, arcs and properties. Rather than having all references to the Employee class simply point to it, as in an object-oriented data structure, text files have to resort to copying the sequence of characters “E”, “m”, “p” etc. into every place that references Employee. The compiler parses and links these all back together into true references later, but the VCS has to cope with the duplication inherent in the string representation.

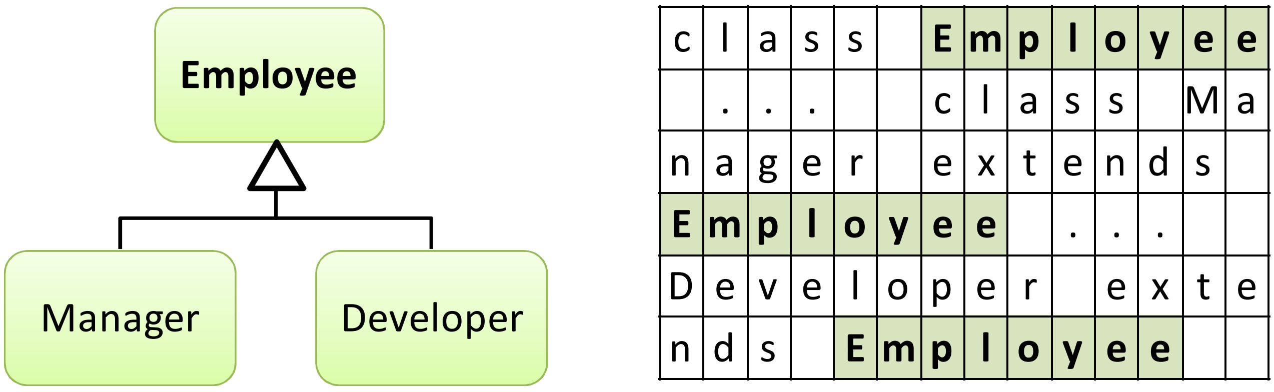

Figure 1: Linking by direct reference vs. by string matching

1.2 Text vs. multi-user editing

Most programming languages store multiple function definitions in a single file, which then forms the unit of granularity for editing and version control. Since the text file has no structure, there is no way to lock just the section containing a single function, allowing other users to work on other functions in the same file in parallel. Even if we know a function addEmployee() starts at character 500 and ends at 600, we cannot lock just that area of the file, because any edits to functions before it may change the start and end positions. Either the whole file must be locked, or else copies must be made for each overlapping editor, and then merged on a character-by-character basis later. The fact that most of these merges can work without problems is largely testimony not to the ingenuity of VCS algorithms, but to the fact that the developers are mostly editing different functions.

1.3 File vs. repository

While the textual format of languages is hard to change because of the inertia of the installed base and supporting tools, modeling languages were free to choose their own data structures for editing and storage. All modeling tools use an in-memory representation that is some kind of graph (although some try and force it into a tree, with ensuing problems). Indeed, even textual IDEs these days often maintain the parsed graph or Abstract Syntax Tree in memory in parallel to the textual representation. Some modeling tools, e.g. Simulink and Rational Rose, chose text files as their native storage format. XML files also fall into this category; the files are less human-readable, but building your own tools to operate on the files is often easier – unless the XML schema is particularly perverse, e.g. XMI.

Moving away from textual storage formats, some tools, e.g. LabVIEW, chose binary files; interaction with the models was only via the tool or its API. Given the complicated structures and interdependencies of model data, this may be a welcome safety net rather than a serious limitation.

File-based representations generally suffer from the problem of granularity: on the one hand, each file already contains too much (e.g. several functions); on the other hand it contains too little: the contents would like to refer to something that is actually in another file. For a single developer and single version, a simple string lookup or include directive can be enough, but across multiple versions this becomes problematic. When created, the reference was to a particular version of a particular element in a particular file, yet the name of that element or file may change in subsequent versions, or another element may be created to replace it. Since the link is across files, and the other versions of that file are locked away in version control, there is no easy way to ensure that the link is maintained. Sometimes this kind of indirection is desired, but when not then the splitting into files imposes it on the developer. The granularity of reference is artificially locked to be the same as the granularity of locking, although it would ideally be significantly larger.

To address this issue, tools turned from files to repositories. A few tools chose relational databases, but relations are a poor fit for graph structures, and tables collect all things of the same type together, rather than allowing modularization into models and sub-models. Some tools chose object databases, which can represent the model graph directly as an object graph. This is certainly the most natural format, and while it is furthest from the traditional textual storage, it is closest to the in-memory structures that developers are used to working with.

1.4 Models vs. version control

Experience has shown that the innate graph nature of models, stronger there than in text files, proves problematic to current version control systems and processes — even if the models are stored as text. Purely textual comparisons of model versions generally reveal little understandable information of changes beyond simple property string modifications. The textual comparison and merge functions of VCSs tend to be useless or plain dangerous when applied to models — hardly surprising, since they were never designed for that task.

In the face of these difficulties, some older or more expensive modeling tools have implemented their own comparison and even versioning functionality within their tool. Nowadays, this solution is becoming less attractive: development processes are strongly tied to the use of a modern VCS, and developers shy away from data in silos in different tools. Other factors also play against it, for instance the patent minefield around model differencing, or the lack of actual success of the comparison tools themselves. Often the results of comparison are shown as a simple tree; more advanced tools or third-party add-ons may also be able to highlight the changed areas of the graphical models themselves.

Another problem for VCSs is that many modeling tools have more than one file for each model. Repository-based tools generally have several, which must be treated together as a single set; since VCSs generally handle only single files, these are normally versioned as zip files. Even file-based tools often have a pair of files, e.g. separating out the conceptual content of the model from its graphical layout. For file-based tools where the modeling language is not fixed —the majority these days — other files describing the metamodel or profile may also be needed for the model files to make sense. Those should of course be versioned separately, but a versioned model file should record the modeling language version it was made with. For less mature modeling tools that are unable to load files made with previous tool versions, the version of the tool itself may also need to be recorded.

2 How mature modeling tool support addresses the requirements

Versioning models in current VCSs is thus something of a nightmare. Some of the problems were apparent even with textual code files, and models exacerbate them; other problems are particular to models. Storing models as text has failed to provide a solution: adoption may seem easier, but actual use reveals the same problems. Worse, the more that models are used like text, the fewer of the benefits of models are achieved: scalability, de-duplication, communication etc.

This may not be such a problem where models are used simply in addition to textual source code, as drawings and documentation. But in Model-Driven Development, where models become the primary source artifacts, more effective solutions are needed, particularly as usage grows. While most MDD tools are in their infancy, there are some mature tools that have been used for 15 years or more, such as LabVIEW, Simulink, and MetaEdit+. All have been used successfully for projects of all sizes and durations, evolving solutions to these questions along the way. MetaEdit+ has the added interest of handling the creation and evolution of modeling languages themselves by client organizations, which is an increasingly important issue. It is also the furthest from text-based formats, using a multi-user object repository, and its range of experience extends more widely to cover the full range from embedded to enterprise applications. While the experiences below will focus on MetaEdit+, this breadth indicates that the lessons learned should be applicable to modeling tools in general.

2.1 Basics of MetaEdit+

MetaEdit+ is a modeling tool that runs on multiple platforms, supports multiple modeling languages (including new ones created by users), multiple representations of the same model (as a diagram, matrix, table or hyperlinked text), and multiple simultaneous users. Models and metamodels are all stored in an object repository, either on the local hard disk for a single user, or on a server for a multi-user repository.

Below we shall look at the overall process of use, and how the three main functions of version control — archive, collaboration and branching —are handled.

2.2 Overall process

While the tool comes with a library of over 50 existing modeling languages, most clients use it for Domain-Specific Modeling [Kel08]: their expert creates a modeling language specific to their problem domain, along with generators producing the appropriate code, and the other users create applications by modeling them with that language. The focus in MetaEdit+ is on ease of language creation and evolution: building the language and generators generally takes 1-3 man-weeks, and full modeling tool support is then available with no additional effort.

For the modeler, there is a strong focus on integration and reuse. Where other tools type the same string into two places to make a link, in MetaEdit+ the link is normally made as a direct reference from one element to another. Where an extra level of indirection is desired, or for links from one repository to another, the link can of course be made by copying the string, and the required lookup will be performed in the generator or during compilation. Objects can be reused across multiple diagrams, and multiple layers of models and modeling languages can be combined to cover all aspects and levels of a system.

The single and multi-user repositories can be used in various combinations to meet the needs of the particular organization and project:

a) multiple single user repositories (e.g. for separate small products, or for modules that will be linked to form a larger product)
b) single multi-user repository
c) hybrid: some single, some multi (e.g. where a single product or module becomes too large a task for a single developer)
d) pairing: two developers on one PC, or sharing a multi-user repository(e.g. high-level UI structure and behaviour by an interaction designer, lower-level details added to the same models by a developer)

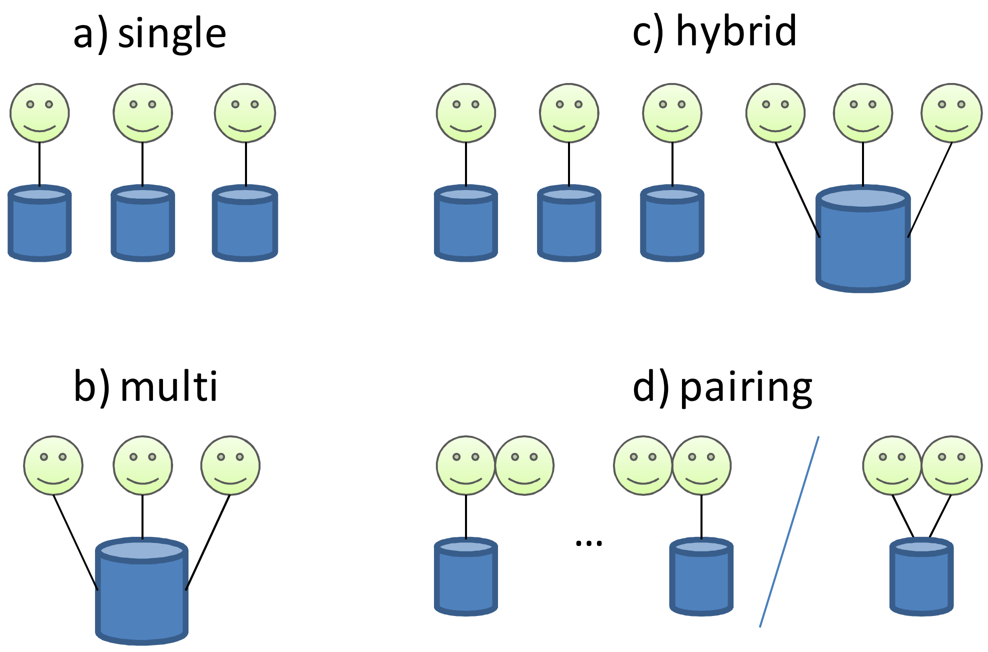

Figure 2: Repository use scenarios

2.3 *Archive*

This works as expected, zips of the repository are stored in whatever VCS you like. This enables you to return to an old version, e.g. for disaster recovery or to make a bug fix to apply to a released version of a product. Choosing versions can be handled manually with the normal OS operations, or then more comfortably from within MetaEdit+, the VCS, or the IDE: MetaEdit+ provides scripting facilities for automating all of these to fit in with a company's own toolset and practice.

For check-in and check-out of whole repositories we advise customers to automate things so the developer doesn't have to worry about the details. A short batch file is called as a macro from the version control system, to zip and unzip the repository in the appropriate directory. In places where security is handled outside of MetaEdit+ (e.g. in the VCS or operating system), the script can go on to log in to the repository, and even open a specific project and graph if desired. This can be integrated with the IDE, so for instance the user could see a tree of repositories, projects and graphs in Eclipse, and double-click a graph to be taken straight to editing in it. These are the kinds of things that developers want: the ease of opening a graph is important, because if it takes too many manual operations, by the time it is open you have forgotten why you wanted to open it in the first place!

One other thing we've found is that with DSM and a multi-user repository, there is no longer a strong need for storing many intermediate versions of individual components: you basically get a "continuous integration" approach for free. Instead of many intermediate versions, it is often enough to version just production and test releases. Even with traditional VCS systems and textual code, it is rare to return an entire component to a state more than one version ago. More normal is to take one small part from the old version, a line or two, and apply it in the current version.

While most issues of comparison and merge will be discussed in the next section on collaboration, comparison is also useful when storing a new version: to help write version comments, and to check that you haven't changed something you didn't mean to. Our customers have had good success at improving model comparison by writing a modeling language specific generator that outputs the model in a human-readable text format. Since they're creating their own graphical languages and code generators for them, it's little trouble to add the simple text generator needed. Each archived version includes the textual summary, so the new version can easily be compared to its predecessor. Unlike textual formats for storage of models, this need not contain every tiny detail, and its format can be optimized for readability and textual comparison rather than ease of conversion to and from a graph.

2.4 Collaboration

Rather than having post hoc merging, or pessimistic locking of entire models, we have fine granularity locking. The information of the graphical layout of a diagram, for example, is locked separately from each object in that diagram, so one person can be neatening up the layout while others are editing individual objects' data. Each object's data is stored and locked separately, right down to individual properties.

We have design transactions that are ACID and long (minutes to hours). We ignore read-write conflicts – developers are used to seeing the latest released version of other developers' work, so it doesn't matter if the owner of that work has updated it in a parallel transaction. Research shows that in design repositories, less than 3 operations per 1000 are write-write conflicts; if the granularity is kept fine, this is thus no problem, but if the granularity is increased to be whole models (as with traditional repository systems or file-based merge systems), there are often changes by two developers within the same unit. (Of course whatever the granularity, there may always be semantic conflicts – object A on one side of the model world may somehow be semantically linked with object B right on the other side, even though there are no such links in the model itself. That tends to indicate a problem in the modeling language, which has failed to capture an important link.)

2.5 Branching

In textual VCSs, a branch is essentially a copy of a file, possibly represented for storage efficiency as a comparison to the base version. In a VCS branch, any part of the file can change in any way, so the tool can offer little real support. In Domain-Specific Modeling, we use Software Product Line Engineering to examine the need for branches – what are the product variants, in what ways can they vary from each other, and which parts are always the same. The commonality and variability are handled separately, so there's no wholesale copying of parts that stay the same. The variability is often handled by creating a modeling language that allows modelers to express it; only kinds of variation legal for that product family can be specified, and the language only covers those points that can vary – a big difference compared with normal "copy" branching. Each product variant has its own model in that variability language, and all product variants share the same commonality (either explicitly as models or implicitly, e.g. in the generators and framework code). A similar approach can be used for managing configuration of components for different releases, e.g. a production release in parallel with a test release.

3 Needs work

Unsurprisingly, an area that still needs work is model comparison and merging. The main needs for comparison and merge in textual version control are solved for models by preventing those needs from arising, with the multi-user version and DSM languages for variability. Indeed, in many ways the solution for models is better than that for text. The need for comparing with the previous version is also handled, with the textual model summary. However, that still leaves some ad hoc cases where it would be convenient to compare and merge two models.

The main problems for ad hoc comparison and merging come when model structures reuse objects significantly, as is common in MetaEdit+. It's not enough to get a model that looks right, with Foo referring to Bar: it must also be the right Foo and right Bar. Obviously, that can be done, e.g. by combining unique internal ids with comparison of property values, but trying to come up with a UI that can make sense of this to the modeler is hard. The wide range of graphical comparison and merging patents held by National Instruments would seem to rule out many approaches; it is an open question how much tools like EMF Compare infringe on them.

Automatic support has been added for one special and rather common case of merge: where there are multiple disconnected repositories, and one modeler makes some "core" models that all others should import, and then link to and reuse in their own models. The core models can be re-exported and imported later, and will update the previously imported ones, without duplicates or damage to the other models.

4 Industry cases

In all the cases below models were edited in MetaEdit+, with the repositories saved to the client's current VCS.

Telecom, fifteen years, hundreds of developers, dozens of sites across the world: Mostly used single user repositories, each with one feature worked on by one developer. This worked well as features were well modularized, and meshed well with existing VCS practices; overall productivity increased by a factor of 10. Where features grew, the multi-user version allowed several users to work on the same repository. Metamodel updates were distributed to each repository, where MetaEdit+ updates the existing models automatically with no data loss.

Finance, insurance products, multi-user, one site: The modeling language and generators were implemented in two weeks to the client's specification, allowing insurance experts rather than Java developers to create the models and generate full code. The tool worked fine, improving productivity by 3-5x, but the modeling language was missing support for variability. That would have enabled them to make a core model of a generic insurance policy, e.g. for buildings, and be able to specify smaller extension or difference models for each actual building insurance policy offered by various companies, rather than copying and changing models.

Medical, configuration, sites on different continents: Direct multi-user repository use across the Atlantic was cumbersome because of the network latency. Much better results were obtained by a remote desktop connection to a client running on the same network as the server: the amount of data involved in mouse movements and screen updates was significantly less than that required for the models themselves. This project also dabbled in saving models as XML, but hit the problems of model versioning and inter-model references mentioned above, and continued with the multi-user repository.

Defence / aerospace, small team on one site: The client had to comply with the FAA requirement that each application be modeled in two different modeling languages. They used FUSION (object-oriented) and SA/SD (structured), with both languages and their models in the same multi-user repository, making the simultaneous parallel development easier. The small team size and lack of experience with databases meant some help was needed from the vendor for major repository administration.

5 Conclusions

Existing version control systems have evolved to offer good support for textual languages. However, that support is not well suited to models, even models stored as text. Storing models as text also introduces a number of problems of its own. By returning to the fundamental principles and functions needed from version control, we can find more appropriate ways of managing the development of multiple integrated models by multiple users. Existing version control systems still have an important role to play in model management, but some functions are best left to the modeling languages, tools and repositories themselves. Direct linking between model elements, multi-user repositories, and domain-specific languages for products and their variation all contribute to the increase in productivity consistently seen with mature DSM tools.

6 References and Links

Alt09 K. Altmanninger, M. Seidl, W. Wimmer, A Survey on Model Versioning Approaches, International Journal of Web Information Systems (IJWIS), Vol. 5 Nr. 3, 2009.

Kel08 S. Kelly, J.-P. Tolvanen, Domain-Specific Modeling, Wiley 2008.