Toys: a Simple Environment for Version and Configuration Management

Giovanni A. Cignoni

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, ITALY
(giovanni@di.unipi.it)

Abstract

Version and configuration management is an important issue in software development. Many solutions have been proposed, but they all fall in two different and far-away categories. From one side, there are basic software engineering tools and best practices, on the other there are complex process based environments that rely on rigorous concepts and protocols.

Toys is a simple environment for version and configuration management that attempts to integrate the process modelling concepts and solutions with the traditional tools and best practices typical of the Unix development environment. This paper presents the rationale behind the definition of the Toys environment and the techniques its prototypical implementation is based on.

1. Introduction

Development of software products is almost completely carried out using computers and software tools. Moreover, all the artefacts of the development process are electronic documents: requirement, specification and design documents, source code, test data and reports, user’s manuals, and so on.

Electronic documents – files, simply speaking – have many advantages. It is easy and economically viable to modify, reproduce, and transmit files. In terms of the software production process, it comes natural to correct errors, to try alternative solutions, to plan the development activities as incremental steps that gradually approximate the required product. Such opportunities are impossible or too much expensive in the production of industrial goods other than software.

However, this bent for change made the software process status ephemeral, continuously in evolution and difficult to be controlled. In other industrial contexts, the concreteness of the products introduces some warranties about the process status.

Because of this problem, version and configuration management is an important issue in software development. Version management is about product evolution with respect to development time and phases. Configuration management deals with the many
files – in their many versions – belonging to a project and, in particular, it faces the problem of selecting consistent sets of files with respect to product properties.

Software engineering proposed many solutions for the management of electronic documents and artefacts in software projects. The research on software process defined high level concepts and protocols to model the software products and the virtual places in which development activities are carried out. These concepts are used to rigorously define and enact the whole software process. On the other hand, in many cases, programming still deals with files and directories and relies on few software engineering tools. Unix plus GNU tools are an example of a development environment that is very poor and primitive if examined with respect to process technology. Indeed, it is adopted by many research and industrial projects and its flexibility overcomes the environments based on more rigorous approaches.

This paper presents Toys, a simple environment for version and configuration management that attempts to integrate process modelling concepts and solutions with traditional tools and best practices typical of the Unix development environment. The former are rigorous and well defined, but often they are too much strict or tied to particular processes, they need complex supporting environments that in many cases are incompatible with other development tools. The latter are flexible, inexpensive and commonly available, but, because based on primitive tools and best practices, they leave the correct implementation of process policies to the responsibility of programmers. Moreover, it is difficult for managers to control a project carried out using the traditional Unix environment. In fact, in such projects there are no managers: they are all programmers, some of them with higher responsibilities. To fill this gap between programmers and managers, Toys defines a model that gives defined names and meanings to the operations the Unix programmers performs on files and directories.

Programmers and managers will benefit from the Toys light and consistent environment. Experienced developers should find Toys a useful collection of tools implementing their best practices and saving menial work. Novices should be able to work productively and without fear of errors. From the management perspective, Toys, without changing the way programmers are used to work, introduces high level concepts and operations that can be adopted in the definition and in the documentation of the software process. This should lead to a better process control by improving the communications between programmers and managers, especially when the latter are not Unix gurus.

The next section discusses the common needs about version and configuration management in the practice of software development. Section 3 describes the basic model of the Toys environment in terms of libraries and operations. Section 4 gives a brief description of the techniques the implementation of Toys is based on. Section 5 compares Toys with other commonly used tools for version and configuration management. Section 6 summarises the results of an experimental use of Toys and states some directions in the evolution of the environment.
2. Version and Configuration Management in Practice

The phrase “version and configuration management” is overloaded of meanings. It is important to define which are the common requirements concerning this issue. We do not want here to specify a full system for version and configuration management; rather we aim to describe which are the usual needs of the community of software developers.

Our first consideration is that version and configuration management is mostly a problem of file management. The product of a software development process is a system made by a set of modules. Several files correspond to each module: source and object files, makefiles, and test data are usual examples. Other project documents, such as requirements, specifications, design diagrams, user’s manuals, and so on, are also implemented by files.

Usually, all these files belong to a defined directory structure that represents the project database. The directory structure generally follows the system architecture, with subdirectories for each module. Additional subdirectories are normally used to store the “other” project files. This is not an over-simplified nor primitive or low-level view of the software development: many projects are carried out in this way and programmers are fully aware of files and directories, especially when the supporting environment is Unix with GNU tools [01, 02].

In order to prevent risks and to optimise the development process, all the files we enumerated must be objects of a wise management. It is possible to recognise four main needs that have to be satisfied by a version and configuration management environment.

- **Version Control.** All files that depend on human activities (typically all source files and documents) have to be versioned. Version control is an effective way to prevent risks, to support old releases of the product and to document the history of the whole development process.

- **Concurrent Development.** To speed up the development process it is important to organise the programmer’s work in parallel. However, to prevent inconsistencies, simultaneous accesses to the same files have to be avoided: lock mechanisms and/or branching ability for subsequent merging are necessary features.

- **Baseline Management.** At any time, a defined version of each file of the project must be available as a reference; the set of such versions identifies a defined configuration of the product that is named baseline. Baselines are generally related to the milestones of the process and must satisfy specified properties. For instance, a baseline represents the starting point of an incremental development step and it must compile and run correctly except for the functionalities yet to be implemented.

- **Simultaneous Variants.** Variants are configurations of the same version of the product made to meet particular requirements as, for instance, the ability to run on specified platforms. Working simultaneously on different variants must not generate conflicts; moreover, all variants have to be kept in sync with the current baseline of the product.
Other typical version and configuration management issues regard product building and variant selection. Product building mainly means efficient recompilation; variant selection refers to automated selection of the files necessary to build a defined variant. These issues are covered by widely known and adopted tools (such as Make [03] and Autoconf [04]), so we do not include them in our list of common needs. The same approach – using known tools, when possible – is also the reason for which, in the Toys implementation, we chose to rely on RCS [05] for basic version control.

Many software engineering tools have been proposed for version and configuration management: from specialised – and by now traditional – shell tools, such as the already cited RCS, Make and Autoconf, to complex environments able to enact and control all the development activities with respect to a predefined process model [06].

Some of the proposed solutions share a common denominator that can be summarised in few concepts concerning libraries\(^1\), i.e. containers used to store artefacts and documents (i.e. files), and operations that it is possible to perform on them to carry out the software process. The project files are stored in repositories, which are public libraries that maintain and make available, under defined protocols, consistent versions of the files. Development activities are carried out in workspaces, which are private libraries that developers use to work on files. Because workspaces are private, temporarily inconsistent files do not affect the work of people outside the workspace. The development process proceeds following a transaction protocol that, in its many variations, is generally known as check-out/check-in. The transaction begins with a copy of the file to be modified from the repository to a workspace; then the development activities take place and, at their end, a new, consistent version of the file is placed in the repository.

---

\(^1\) In this context, the term “library” does not have the usual meaning of a file that collects binary objects.
Figure 1 represents the repository/workspace model and the dynamics of the check-out/check-in protocol. For a more complete description of this framework, we refer to [07], where these concepts are applied to all the objects involved in the enactment of a software process. A discussion of the specialisation of this general framework in the many approaches to process modelling and enactment can be found in [06].

In the practice of software development, while project documents and artefacts are files, repositories and workspace are directories that programmers are used to create, remove, navigate and work in. The traditional Unix programming environment provides few yet very flexible tools that developers use to build by hand their repositories and workspaces. It is important to note that programmers often ignore the repository and the workspace concepts: they simply follow their needs.

For instance, Lndir [08] replicates a directory structure in which files are substituted with links to the original ones in source directories. Originally, Lndir was provided to support installation of different variants without wasting of disk space. Lndir – or one of its shell substitutes – is also usually exploited to realise workspaces in which links save space and ensure synchronisation with the contents of a reference directory that implements the project repository.

The use of Lndir is just an example; experienced programmers know many tricks about the combined use of the Unix shell commands and utilities with the basic software engineering tools such as RCS and Make. This huge compendium, generally lacking of documentation and orally handed down, makes up a very smart and efficient developing environment which power lies on its high adaptability to the needs of every software process. Unfortunately, such environment strongly relies on the experience of the programmers: they have to know the tricks, they have to recognise when it is useful to apply them, and they must avoid errors while issuing complex shell commands.

3. Toys Model: Libraries and Operations

Toys is a simple environment for version and configuration management that aims to integrate process modelling concepts with practical solutions to the common developer’s needs described in Section 2. Toys defines a model that gives a name and a meaning to the operations that programmers are used to perform on files and directories. This makes possible to describe and to control the development process without coping with the Unix tricks and tips.

The Toys model is based on three kinds of libraries and a small set of operations. Libraries are an implementation of the concepts of repositories and workspaces; operations are derived by a general transaction model that, at its lowest level, coincides with the check-out/check-in model. Toys libraries are described in the following.

- **Project Repository** (PR). It is the controlled library where all the versions of all the project files are maintained. The PR is unique and it is hierarchically structured following the architecture of the software system being developed. The PR is not a development space (other Toys libraries are suited for such activities) but the library that stores the process products and makes their consistent versions publicly
available in the project. In particular, the PR by default publishes only the versions belonging to the current baseline.

- **Work Tree** (WT). It is a copy of the structure and contents of the PR, created to perform development activities. Many WT’s may exist simultaneously during a software process, their contents are kept in sync with the current baseline: all updates to the baseline are automatically reproduced in all the WT’s. A WT is a read-only library, it cannot be directly used to modify the product, rather it is devised to support activities that need separate and independent working areas yet constantly aligned with the current baseline.

- **Work Space** (WS). It is a portion of the hierarchical structure of a WT, that corresponds to a module and in which it is possible to carry out development activities that modify the product. All intermediate versions of files modified during such activities are stored in the PR, but they remain local to the WT the WS belongs to and do not affect the published baseline.

The creation of a Toys library generally corresponds to the beginning of a development task. For instance, the creation of a WT is necessary to build the product and to perform system testing: if there are more variants of the products it is possible to test them simultaneously using a dedicated WT for each of them. To perform true development tasks a WT is necessary: one or more WS’s are opened in the WT for the modules to be modified. At the end of a bug fixing, for instance, all the WS’s used to modify the modules affected by the bug are closed: it is this action that updates the baseline and makes the fixed version of the software system available to all the WT’s. The relationship between modules and WS’s reflects the idea that the modular architecture of a software system is also a practical way to organise the development process in small tasks.

The Toys operation set is defined according to this approach: for each kind of libraries there are creation and deletion operations that correspond to the begin and the end of typical developing tasks. To complete the set, check-out and check-in operations are provided to extend the transaction model down to the file level.

- **Create a PR.** This operation corresponds to the beginning of the software process. It creates a PR that consists only of the empty root module (that also represents the first baseline).

- **Create a WT.** This operation creates a new WT that replicates the contents of the PR as defined by the current baseline. The new WT must have a unique name that identifies it in the project. This operation corresponds to the beginning of each major development task that needs a separate work area.

- **Open a WS.** This operation opens for modification an identified portion of the WT structure, i.e. a module. Opening a WS is a necessary operation at the beginning of each development task that requires modifications on a module. Modifications are local to the WS and do not affect the baseline.

- **Check-out a file.** This operation reserves a file in a WS for modification. The reservation has effect on all the WS’s currently opened on the same module in other
WT’s. File editing is the most refined level in the identification of development tasks: this operation begins such transactions.

- **Check-in a file.** This operation ends the file modification transaction. Reservation for modification is released and a new version of the file is generated. The new version is stored in the PR but it is not made publicly available, i.e. the baseline is not updated. However, the new version is available in the WS’s opened on the same module in others WT’s.

- **Close a WS.** This operation revokes the modification rights on a WS, which becomes again a simple WT portion. All performed modifications are transferred to the PR and the baseline is updated. The operation is used when the development task on one module is over; it fails without any effects if the WS still contains some checked-out files.

- **Delete a WT.** This operation actually destroys the WT. The operation is to be used at the end of each major task that began with a WT creation. The operation fails without effect if the WT contains one or more opened WS’s.

- **Delete a PR.** More formal than practical, this operation terminates the software process. No WT’s must exist and no more WT’s can be subsequently created from the resulting “hibernated” PR (we have some hesitation to actually destroy a PR: hibernation makes possible to re-create a PR from a previously frozen one).

Other minor operations involve WS’s and regard creation and deletion of files and modules. Except the root module created with the PR, all modules are to be created as sub-modules. Because this is a modification of the product architecture, the operation must be issued in a WS. The same happens for similar operations on files. As usual, closing the WS propagates the effects of creation and deletion operations to the baseline and to all WT’s.

### 4. Implementation

High level concepts like repository and workspace were the reference items in the definition of the Toys model. On the implementation side, a specific goal of Toys is the explicit use of the traditional and very reliable Unix environment and GNU tools.

For explicit use of the Unix environment we mean that Toys does not introduce any abstraction layer above the file system: directories are directories and files are files. Many development tools are and have to be used at this level (Make and Autoconf, for instance). Introducing an abstraction layer may be simply useless, because it forces programmers to work at two different levels, or harmful because of the impossibility to use such – powerful – tools.

For use of Unix and GNU tools we mean compatibility and synergy with widely known and adopted tools. Because many programmers use RCS, Toys uses RCS for file check-out and check-in. Moreover, the way Toys exploits RCS is compatible with that
used by GNU Emacs [09] in its Version Control interface, so Toys users are encouraged to use Emacs and Emacs users will find no problems adopting Toys.

In practice, Toys is a set of shell commands corresponding to the operations defined in Section 3. These operations work on the directories that are used for software development. Their results, in terms of how directories and their contents are modified, reflect the same ideas and principles the common programmer’s best practices (or tricks) are based on. In the following, we describe how Toys libraries are implemented. For a graphical representation of the structure of PR’s, WT’s, and WS’s we refer to Figure 2.

- A PR is a directory tree identified by the name of the root directory. To each module of the software system corresponds a sub-directory and there is no limitation in sub-directory levels. In each directory of the tree there is a RCS/ directory used by RCS to store file versions. While in the RCS/ directory are stored all the versions of the module files, in the module directory, for each file, there is only the version that belongs to the current baseline. These versions are read-only and they are the ones referred as links in the creation of WT’s.

- A WT is a copy of the directory structure of the PR in which files are substituted by links to the files in the PR. Directory names are maintained with the exception of the name of the root directory: this must be different and identifies the WT. Moreover, the copy process excludes the RCS/ directories. The use of links saves space

---

Fig 2: Implementation of WT’s and WS’s
and keeps WT contents in sync with the baseline. The absence of RCS directories makes impossible to accidentally check-out files in a WT.

- A WS is a modified WT directory: links are substituted with true files (a copy of the ones in the PR) and a RCS link is created to connect the WS with the RCS directory in the PR. This solution makes RCS able to globally prevent the simultaneous access to the same file.

Many experienced programmers are able to manually manage similar directory structures exploiting tools such as Lndir or issuing sophisticated shell command. Toys operations are a mean to do such work automatically.

The use of links may lead to some problems. For instance, it is possible to modify a file in the PR just by editing the corresponding link in a WT. However, first of all this action needs to overcome the read-only permissions on the file in the PR, then smart editors, such as Emacs, warn the user about opening files via links. Moreover, even if such thing happens, because the RCS directory in the PR is not linked, the modification does not affect the true repository of the project. In general we found that it is very difficult to accidentally misuse or damage the contents of Toys libraries. Indeed, the integration of this implementation of workspace and repositories with other commonly adopted development tools is very smooth and efficient.

The current Toys prototypal implementation does not exactly match the model described in Section 3. For instance, because the PR is just a directory tree, there are no operations for its creation and deletion. The most remarkable difference regards WT synchronisation with the baseline. At the current state, with respect to addition and deletion of new files and modules, WT’s are not kept in sync automatically but upon explicit request. In practice, this solution does not appear as a serious lack. The upgrade of the baseline is an important and well known event in the software process, so it is realistic that WT managers are aware of it to manually request the synchronisation. Moreover, an additional command was introduced to interrogate the status of the Toys libraries having information about WT’s, opened WS’s, checked-out files, and baseline versions.

5. Related Work

Compatibility was the most important issue in the design and realisation of Toys. Our goal was a smooth integration with the Unix traditional programming environment. In this perspective, compatibility regards both development tools and developer’s usual practices. The freely distributed source trees of the many GNU tools are an example of the “way of programming” we refer [10]. Besides the GNU and other similar free software projects, many research, commercial and educational projects are carried out following this way.

In this context, RCS is the most widely used tool for version control. But RCS covers only the first two key-points of the common needs for version and configuration management listed in Section 2. RCS users, however, can easily adopt Toys, because Toys relies on RCS and does not make any limitation in its use. From this point of view,
it is possible to say that Toys simply adds baseline management and simultaneous working spaces to those contexts in which RCS was already adopted.

CVS [11] is the existing configuration and management tool that is most similar to Toys. Toys shares with CVS the workspace and repository approach and the relationship between system modules and working directories. However, CVS working directories are based on copy; this means that there is no control about concurrent modification on files. CVS performs version control only locally to the working directory; consequently, subsequent merge is a mandatory solution in the case of simultaneous modifications. We think that the lock vs. merge question concerns process policies and then the supporting environment has to support both strategies. Toys.

6. Conclusions and future work

The main goal in the definition and the implementation of Toys was simplicity. By experience, we believe that complex models are difficult to be translated in useful development environments. Attempts in this way resulted in many open questions [12], many of which concerns the integration with the real world development tools.

On the other hand, Unix and GNU tools that are widely adopted by programmers date from ten or more years ago. Even if new and improved versions are continuously released, their general formulation is the same, nor it is changed the impact they have on the software process. Moreover, there is a gap between the use of the tools by developers and the manager's vision of the software process.

Toys does what managers want in the way Unix programmers do. The Toys model, while concise, can be referred in the description and in the documentation of the development activities. The Toys implementation does not drastically change the traditional Unix environment the programmers are used to.

The current Toys implementation is adopted for Toys maintenance and further developing. The most relevant case of use, however, is an educational project carried out during a course of software engineering held at the University of Pisa, involving 25 students distributed in 5 development teams.

The subject of the project was an incremental development step performed following the PSS05 [13, 14] process directives. The software system being developed, yet not large, was significant in size and architecture: it counted 10 modules and more than 40 C++ source files; compilation was driven by a two level hierarchy of makefiles. From the point of view of Toys usability it is important to note that the students were novice developers and the instructors, acting as project managers, were not Unix gurus. The instructors used the Toys model to tailor the configuration management activities defined by the standard. Students, using such documentation and the Toys environment, were able to complete their tasks without significant problems or delays.

Regarding the evolution of the Toys model, we plan to investigate the PR-WT synchronisation problem. As described in Section 4, the on-request solution currently implemented differs by the automated synchronisation established by the model. In some cases however, it seems useful to have uncoupled WT’s, for instance to prevent un-
wanted and unexpected changes in the software system. Offering both behaviours will surely improve the flexibility of the Toys environment. Other work will include the porting in C/C++ of the Toys commands, currently implemented as shell script, and the realisation of a graphical interface for the management of the Toys libraries.

7. Acknowledgements

Many suggestions about the Toys model, often expressed as requirements, came from Carlo Montangero, that also held the course in which Toys was used as development environment. Vincenzo Ambriola helped in the revision of the paper. The CAMPUS project partially founded the implementation of Toys.

8. References


Toys is distributed as free software at http://buzz.di.unipi.it/Toys/.