Transactional Upgrade Rollback: the DSL approach

Work Packages 3 and 2

Nature : Technical Report 004
Version : 1.01
Date : January 4, 2010

Transactional Upgrade Rollback: the DSL approach

John Thomson,* Paulo Trezentos

December, 2009

Abstract

Todays mechanisms of providing transactional upgrades and roll-backs on Linux systems are incomplete and unsafe. They do not provide the possibility of returning to a previous system state or, when they do, they don't predict if the roll-back is possible before execution. In this article we propose a new approach based on the use of a Domain Specific Language that formally allows one to know beforehand if the rollback is possible or not. Since the DSL approach requires a significant number of conditions to be met, we devise mechanisms to overcome the absence of these requirements and to minimise the number of cases where roll-back is infeasible.[3]

*Vasco Silva, Andre Guerreiro

1 Introduction

In Free and Open Source Systems (FOSS), there exists a distributed mechanism for developing and releasing software packages that is non-centralised and it is this mechanism that allows for development in all areas of the components of these systems. A package is seen as the smallest collection of software that allow a new set of features to be performed by a machine on which it is installed. They are normally archive files that contain the source or binary files necessary for a particular architecture of system to compile or run the software. Each package normally performs a single role and performs it well. Using the ideas behind Object Orientated Programming, the idea of packages is to promote code re-use, to create efficient pieces of software that interact with other functional blocks and to have software that is easily updatable by the end user that can be distributed over a variety of mechanisms. One of the problems with a distributed code-base such as that present in GNU/Linux and other FOSS based systems is that error checking and collaboration becomes much more difficult tasks. Checking for consistency not only with the packages that are part of the system but also with the package universe becomes a very difficult problem. Traditional helper utilities that assist package maintainers normally check for the usage of shared libraries and for conflicts and dependencies that certain packages will have with other software that is available. This information is normally produced by helper utilities and package maintainers to identify what will and what won't work with a developed package. This information is normally stored as meta-data in the package as a maintainer (or configuration) script file and used by package meta-installers to check and solve dependency trees for conflicts and warnings. A meta-installer apart from analysing some of the meta-data will also pass package information to the actual installer such that they can perform the operation of installation. The meta-installers also are used for generating installation plans as well as solving the satisfiability problems. One aspect that has not been adequately researched and implemented by the meta-installers according to D3.1 and D3.2 is that of transactional roll-backs. Roll-back is the term used to describe when a software package is reverted to that of a previous version. It is the logical inverse of an upgrade and perhaps because it is seldom used, it has not been investigated nearly as much as the upgrade problem. As part of the MANCOOSI project, Caixa Mágica Software are looking at designing, creating and implementing a new roll-back mechanism for package management using technologies previously researched. CMS have already implemented a roll-back like feature once before for the branch of APT that they use, APT-RPM. Using a Domain Specific Language (DSL) that has been developed as part of Work Package 3 and specified in Deliverable 3.2, CMS intend to create a fully-fledge roll-back mechanism for use with their package meta-installer as a proof of concept of the MANCOOSI research. CMS already had a feature whereby a roll-back could be called as a paramter from APT-RPM and the current version of a package would be uninstalled and a named, prior version would be installed. This is not actually equivalent to roll-back but is a tidy mechanism for installing an older variant

of a package.

$$\text{Upgrade version } \vec{XY} = \text{Install version Y + Remove version X}$$
$$\text{Uninstall version Y + install version X} \neq \text{Roll-back}(\vec{YX})$$
$$\text{Roll-back } (\vec{YX}) = (\text{Upgrade } (\vec{XY}))^{-1}$$

There are many issues with roll-back that need to be addressed and in the course of this document we aim to suggest what could be possible solutions to the issues raised and how we can use existing state-of-the-art technologies to focus our research.

Packages Abstracting over format-specific details, a package is a bundle of 3 main parts:[3]

Package
- 1. Set of files
  - 1.1. Configuration files
- 2. Set of valued meta-information
  - 2.1. Inter-package relationships
- 3. Executable configuration scripts

These components can be used by the Package Meta-Installer and Installer on the computer system to generate software that is customised to the environment and availability of features on certain machines. Another approach that is discussed later is that of provisioning and automated tools to help realise this. Briefly, provisioning allows system administrators to efficiently set up a large number of systems without having to repeat a lot of similar configuration settings on each machine.
Mathematical definitions:

1.1 Commutative:

[6] This is a mathematical property that means if the elements of something are re-ordered that we do not change the end result. $A + B = B + A$

1.2 Deterministic:

[7] No elements of randomness are involved in the propagation of the state of a system from one state to another. All inputs are therefore, if isolated from any other systems, combined to perform a set of outputs. It may not be possible to capture the method or algorithm in which the inputs match to the outputs but if a sufficient number of input/output combinations are captured it may be possible to model the cause and effect of systems. Also if the system is not isolated sufficiently from another system it might be that although the smaller system is non-deterministic that if the larger system is modelled sufficiently that the larger system is deterministic. In computing, hardware failures and human inputs can be modelled as

non-deterministic inputs to the system. Petri-Net model of computational systems http://en.wikipedia.org/wiki/Petri_net

1.3 Irreversible functions:

[8] If P $\neq$ NP holds true then there may be irreversible functions.

1.4 Trap-door functions:

[9] Trap-door functions are a one-way function that relies on a secret piece of information. If the information is lost or not stored it is very difficult to recover the original information. Such a problem occurs when we try to analyse the state of a system without any prior knowledge and try to deduce how the system arrived at this state. Conditional commands are an example of a such a function, where if we know how the state maps into another and have sufficient inputs and the current state of the system that we can calculate the rest of the inputs.

```
f(A,B):= AxB; f(4,?) = 12; ? = 3;
```

If however we only have the answer we can not determine what the possible inputs are.
By using a combination of capturing the state of the system, using a representation of the system in terms of a model and by analysing the real data and the model we hope to provide a mechanism for transactional roll-backs.

2 State of the art

We can describe the state of the art of existing tools in terms of scope and properties.
The scope can be analysed with respect to two different criteria:

Time : For how long a specific upgrade can be undone? During the transaction, until the next transaction take effect, for N days, forever.

Granularity : what is in the scope of the roll-back mechanism? File system, RPM files, single file.

Where each analysed tool can be classified in terms of scope:

Rollback tool	Granularity	Reported time period
NexentaOS	All file system	Since last snapshot
Conary	Set of files contained in a package	Triggered by the upgrade of the package
NixOS	Set of files contained in a package	Permanent
MacOS Time machine	All file system	Since last snapshot. Triggered by user, time or changes.
Apt-RPM	Set of files contained in a package	Triggered by the upgrade of the package
Pacman-ARM	Set of file contained in a package	Daily
Btrfs	All file system	Since last snapshot.
~~Zumastor~~	~~Filesystem~~	~~Since last snapshot~~

Table 1: Classification of rollback tools regarding scope

And we can classify in terms of properties:

Out of order : rollback can take place in an out of sequence manner;

Merging configurations : if a file was modified after the transaction, the user modifications are merged into the previous configuration;

ACID properties : respect Atomicity, Consistency, Isolation and Durability;

File system independence : the roll-back mechanism don't depend on a specific file system;

User transparency : user doesn't need to perform special operations to roll-back (as reboot or other intrusive operations);

In terms of configuration management there are a few mechanisms that are capable of provisionning. With this type of system, the system state is stored and can be restored. Rather than focusing on having individual packages state's saved, provisionning enables the entire configuration state of the system to be captured and to

Tool	Out of order	Merging	ACID	FS independence	user transparency
NexentaOS	Yes	No	Yes	No	No
Conary	No	Yes	Not sure	No	Yes
NixOS	No	Yes	No	No	Yes
Current Apt-RPM	No	No	No	Yes	Yes
Pacman-Arm	Yes	No	Not sure	No	Yes
Btrfs	Yes	No	No	No	No
~~Zumastor~~					

Table 2: Classification of rollback tools regarding properties

be recalled at a later stage. This is useful for large enterprises where working on rolling back individual packages is seen as too time consuming and it is felt that restoring the system to a known state is more important. This fire-and-forget type of configuration management is certainly easier for administrators but configuration settings that have changed from the state of provisionning may be lost. Using the DSL it may be possible to save a configuration state of the system and use algorithms to detect which DSL statements need to be run to get a system to that state from the current state. For more information on Fedora's implementation of such a mechanism http://www.redhat.com/rhn/rhntour/. For other implementations it is possible to see Puppet http://reductivelabs.com/trac/puppet/ which runs on top of Augeas http://augeas.net/. There are other mechanisms and applications that bridge between configuration and package management. For a definition of provisioning this might be helpful http://help.sap.com/saphelp_nwidmic71/en/mc/dse_provisioning.htm.
Conary cannot be used out of order http://lwn.net/Articles/138358/. Using absolute changesets though it should be possible. Keeping all the possible combinations of changesets as absolute change-sets appears to be the limitation to achieving this.
http://blog.chris.tylers.info/index.php?/archives/17-How-to-Rollback-Pack
html

2.1 Nexenta OS

Nexenta is a combination of OpenSolaris, the GNU utilities, and Ubuntu. It is a free and open source operating system based on Nexenta Core Platform 1.0 (NexentaCore includes complete OpenSolaris kernel and runtime) using GNU applications. This operating system uses a specific filesystem – ZFS. The ZFS filesystem inserts a lot of features that have the capabilities of the filesystems becoming administrative control points. This will allow: snapshots, compression, backups, privileges, etc. The application that makes available these options to the user is Apt-clone. Apt-clone uses capabilities of ZFS commands and apt-get, with its own features. It uses snapshot's capability to manage the system and creates one checkpoint every time a user upgrades a package. Once the installation done and working (live), one

can proceed to the rollback. Apt-clone manages Grub menu and ZFS system pool filesystems.

2.2 Conary

Conary is a distributed software management system developed by rPath. Conary has the same objective as dpkg and rpm, but is more than a Package Management System, it can also be a package creator, a repository of software, and a versioning tool. The tool to manage the installations, upgrades, and rollbacks is named Conary. Like installer, it has an interesting difference from the other ones: Conary, after downloading and first installing a package, updates faster than other tools because it only requires downloads to update a file and does not require the full binary. It has the capability of doing rollback. Every time a user installs a package that creates a rollback point, the tool takes a snapshot. Conary was developed by the rPath community and is implemented as the base system of managing packages of the rPath Linux. A Linux Distribution based on rPath Linux is a Conary-based system. Foresight Linux is based on rPath Linux, it is thus a conary system-based and they use all the functionalities of conary to manage packages.

2.3 Nix OS

NixOS started around 2004 with a PhD Thesis, as an experience to use a purely functional way. The intention is to solve package dependency problems. This experiment aims to see if is possible an operating system in which software packages, configuration files, boot scripts and the like are all managed in a purely functional way, that is, they are all built by deterministic functions and they never change after they have been built. The main characteristics of Nix package manager(Package manager used on NixOS) is:

- The entire system — kernel, system services, configuration files, etc. — is built by a Nix expression in a deterministic and repeatable way.
- Since configuration changes are non-destructive (they don't overwrite existing files), you can easily roll back to a previous configuration.
- Upgrading a configuration is as safe as installing from scratch, since the realisation of a configuration is not stateful. This is a result of being purely functional.
- Multi-user package management — any user can install software through the same mechanisms that the administrator uses. This is not the case for most package managers such as RPM.

2.4 Apt-RPM

Apt-RPM is a meta-installer that selects, retrieves and installs RPM packages. Apt-RPM is a fork project of APT. The original APT (Advanced Packaging Tool)

project was initially developed to extend some of the capabilities of the dpkg command, which allows the installation of local DEB files. The DEB files contain applications to be installed. The apt comes from the lack of features at the installer level(rpm/deb) that are important to the user, such as Retrieving files, solving dependencies or searching for packages. Rpm has a rollback feature that allows a backup of all files of the version that has to be replaced to be stored in the disk in each transaction. This files are kept in a RPM package associated with TID (transaction ID). This feature is very interesting but has two major problems: a) a large space is needed in the disk since all files, even binaries, are maintained and b) changes performed by installation scripts outside of the set of RPMs are not stored. An evolution of this approach was proposed in 2005 by CaixaMágica Software in the framework of the EDOS FP6 project The most interesting in this scope is transactional rollback at meta-installer level. Like Apt-rpm exists other meta-installer using rpm-rollback feature. An example of this is yum to confirm this, we can check that the version of rpmlib is the same for both:

```
[root@localhost~]#rpm-qRapt
  rpmlib(PayloadIsLzma) <= 4.4.6-1
[root@localhost~]#rpm-qRyum
  rpmlib(PayloadIsLzma) <= 4.4.6-1
```

2.5 Pacman-ARM

Pacman is a utility which manages software packages in Linux. It uses simple compressed files as a package format, and maintains a text-based package database (more of a hierarchy), just in case some hand tweaking is necessary. Pacman, like previous meta-installers, will add, remove and upgrade packages in the system, and it will allow to query the package database for installed packages, files and owners. It also attempts to handle dependencies automatically and can download packages from a remote server. Pacman allows the user to perform rollbacks. To do this, Pacman uses Arch Rollback Machine (ARM). Arch contains archived snapshots of all the repositories going back to 1 November 2009. The rollback consists in installing the previous version that is available in repository.

2.6 Augeas - a configuration API

Augeas is not a tool to manage Rollbacks (because has not features to upgrade packages), but can be useful for the rollback on the level of configuration files. Augeas is a configuration editing tool. It parses configuration files in their native formats and transforms them into a tree. Configuration changes are made by manipulating this tree and saving it back into native config files. Thus, it can be used to save the configurations before the upgrade of a package. If a rollback were needed, the configurations could be reverted again using this tool, to load previous definitions and set on the specific configuration files. Puppet is an example tool

using Augeas. Puppet is a tool to make automated system administration, it's used, among other things, for deploy central configurations for clients. Puppet uses the Augeas features to distribute and load these configurations.

2.7 Btrfs filesystem

Btrfs is a new copy on write filesystem for Linux aimed at implementing advanced features while focusing on fault tolerance, repair and easy administration. Initially developed by Oracle, Btrfs is licensed under the GPL and open for contribution from anyone. Btrfs is capable of creating lightweight filesystem snapshots that can be mounted (and booted into) selectively. The created snapshots are copy-on-write snapshots, so there is no file duplication overhead involved for files that do not change between snapshots. A snapshot can be created at any time, it should do before a upgrade, or on any moment that user wants. A rollback to an older snapshot is not destructive to data. It switches to an earlier snapshot, and later snapshots are still available afterwards. The user could choose which snapshot will be mounted next, and making that choice does not affect or destroy any other snapshots.

2.8 Zumastor

Zumastor has not been active since the middle of 2008 when DSL 3.1 was created. No other projects seem to refer to it or implement their findings. This may be for many reasons and the project may yet produce notable results but at this moment we will consider other package management systems.

3 Domain Specific Language Intro/Re-cap

As part of Work-Package's 2[4] and 3[1][5], a Domain Specific Language (DSL) was decided upon as a potential mechanism for realising the aim of transactionally protected package upgrades and roll-back. The DSL has been defined in Deliverable 3.2 but it is a language that has been designed to be extended and as such in this document we will refer to version 1 as corresponds to the first release of the DSL. For our proposed solution we are going to use an abstraction of the system state in terms of a model of the system. This model is defined as per Work-Package 2. The important part of our implementation in Work Package 3 is to realise that there is a model of the system and that we can modify the state of the variables of the system by using transformations in terms of the ATLAS Transformation Language (ATL)[5]. Through monitoring the configuration states and by describing the action of maintainer script files contained in package configuration files, we can capture the modification of states using our model and the DSL. Erroneous states of the model will be detected by the simulator and failure detector and once this erroneous flag has been reported back to our sub-system we will provide a roll-back mechanism using the current state of the system, the captured configuration states and the transaction logs to revert packages back to a previous version. We also enable the end user to select as a switch to apt-rpm the ability to roll-back transactions. The user will only be presented the sub-list of roll-back enabled transactions from the entire transaction state. The DSL has been designed to capture particular failures that occur with package upgrades that currently occur and also to present new facilities allowing roll-back but generally providing a transactionally protected package management. The DSL by definition is not a fully Turing complete language but rather an extensible language that has been designed to capture the effects of maintainer script files being run on systems. Using the abstraction we will be able to detect certain known failures before the packages are even installed and inform the user that the system will not work as expected. The grammar of the DSL can be represented as a tree structure as show in Figure 1. The tree structure can have recursive elements, hence be cyclic and potentially be infinite in size. As the leaf nodes are individual DSL statements or an empty statement, as long as the maintainer scripts from which the DSL is being formed are not recursive, the DSL grammar tree will be finite in length. If there is no limitation on the database size then the leaves can be pruned and formed into a sequential list. The ordering of the list is important as the DSL statements will be performed in that order on the model and should mirror those of the real system as much as possible. Also in terms of roll-back the order will be used to generate the roll-back equivalent. The system and the changes that are performed on it that modify the state are not necessarily commutative. This means that we cannot guarantee if we perform one operation first and then the second that it will lead to the same result as doing the operations in another order.

DSLiD corresponds to a unique auto-incrementing key entry for the log. Every upgrade of all types, including roll-back and even downgrades, will be stored as

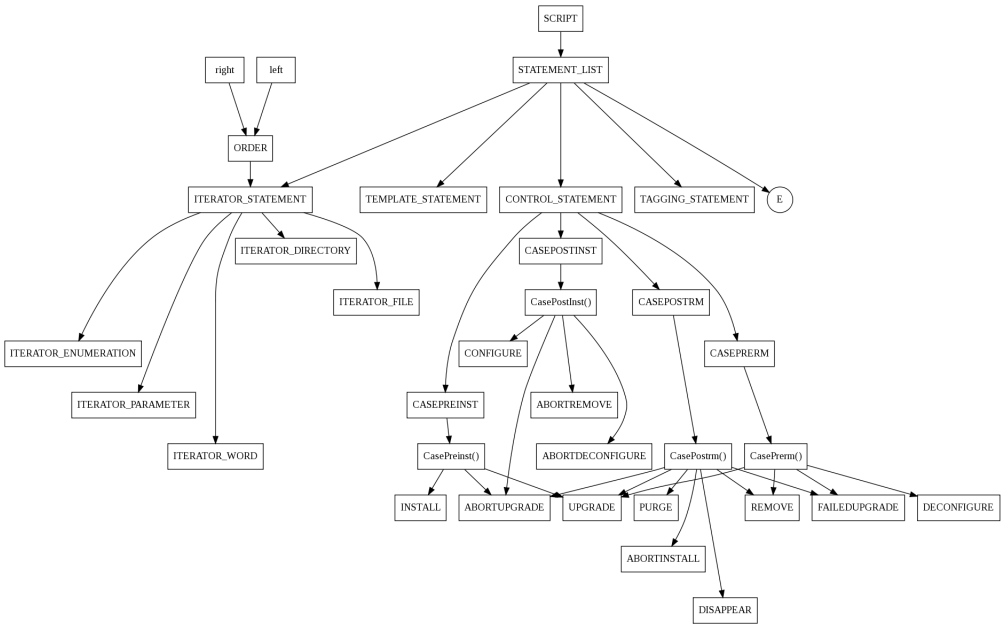

Figure 1: A Tree representation of the DSL grammar

DSL elements to store in log

DSLiD	TransactionID	DSL_Statement	DSL_ParentID	DSL_InvCommand	Dependent
1	1	ChgGrp(sys, file)	1	ChgGrp(prior, file)	No
2	1	ChgGrp(sys, file2)	1	ChgGrp(prior2, file2)	No
.	.	...	.	...	.
.	.	...	.	...	.
.	.	...	.	...	.
x	1	ChgGrp(sys, file(x))	1	ChgGrp(prior(x), file(x))	No

Table 3: Sample DSL Log

an entry, corresponding to every time a DSL statement is run. If an upgrade fails this log will be used to return to the original system state. The most recent DSL statement will correspond to the latest DSLiD. Older events are therefore stored as 'lower' entries and branches in system state are serialised by capturing every statement.

TransactionID refers to a group of DSLiD statements that correspond with Package scriptlets. Many DSLiD statements will correspond to a single TransactionID but each DSL statement is associated with a single transaction. Rolling back a single DSLiD statement would not revert all the changes made in a package scriptlet and so would leave the package in an intermediate state. Even though it may be possible to roll-back individual DSL statements we want to be able to roll-back at the granularity of packages. We can however use the DSLiD statements to search and filter Transactions at a lower level to identify when a particular action was

performed.
The DSL_Statement corresponds to nodes as defined in Figure 1 and the syntactical elements of the DSL that can be found in the specification of the language, for version 1 namely in Deliverable 3.2[5]. The statement will capture the variables that are modified and the parameters that they are passed.
DSL_ParentID is a method for linking TransactionIDs into a larger transaction. For instance if a set of packages are all upgraded as a command by the user, that larger statement can be broken down into the upgrade of individual packages. New generated ParentIDs will be unique but they may be common to multiple transactions. Each TransactionID that corresponds to the same ParentID are the top-most level in a transaction from which sub-transactions are generated. Sub-transactions allow for individual packages to have corresponding DSL statements. For a parent transaction to be capable of roll-back it is necessary to meet the condition that all the children of it have roll-back capable statements associated. Any element incapable of roll-back will mean that the parent transaction cannot use roll-back.
The DSL_InvCommand is a location where the inverse operation of the DSL can be stored or if it refers to a set of DSL statements it may refer to a TransactionID. If a command is not stated, found or cannot otherwise be generated then it is assumed that for this DSL statement there is no roll-back command and hence none of the parent transactions can be used for roll-back.
Dependent is currently a boolean flag or it could be associated with another TransactionID to state that another transaction has to be capable of roll-back for this one to be able to roll-back. There may be no need for this flag or key association.

4 Requirements for a roll-back component

For roll-back we require a certain number of features and there are some that are desirable. A component diagram of what is required to develop a roll-back mechanism using the DSL is shown in Figure 2. Table 4 makes reference to some of the properties required by a roll-back system using a DSL. What is key is knowing which package from the source configuration is desired to have a roll-back operation performed on it. Also the target configuration, or the mechanism to provide a roll-back is required. At this stage we have to identify which files and environmental variables need to be modified to restore the system to the previous state. Mechanisms such as snapshots which utilise copy-on-write schemes, store the state of the whole file system (depending on the granularity) and if implemented correctly allow the system and importantly the package to be reverted back to a particular saved state. A simplistic roll-back will not need the state of the current system as it will simply revert back to a previously installed backup of the package or it will even more naively just install the previous version of the software as identified by the target identifier. As stated before this is not really roll-back but a removal of the previous software and an installation of the older package, bundled together in a neat mechanism. Where roll-back differs is how the changes since the installation of the upgraded version are merged back. Most solutions will either use the configuration file of the old package or if they are more advanced they might save the configuration on upgrade and restore it on roll-back. A further advancement is that of using the old configuration and merging changes back.

Roll-back configuration files

What to do with the config files	Mechanism	Benefits	Disadvantages
Use config from target package	Overwrite	Simple	Loses all user changes
Use a backup configuration	Restore old file	Fairly simple	Loses changes since update
Use a hybrid config	VCS merge target and source	Keeps user changes	Complex
User choice of merge	VCS + user selection	User responsible for changes	Potentially complex for user
Use source configuration	Overwrite	Simple	May not work with old binaries

Table 4: How to manage the configuration files on roll-back

The Log system and mechanism is critical to the design of the roll-back mechanism as we will be using it to retrieve the inverse DSL statements and decide on a plan to get from one configuration state to another. Having a reliable log is necessary if we are going to be able to determine the state of the system and move between states. The design of the logging system should therefore be ACID compliant to ensure that transactions between configuration states are maintained. Of the types of the

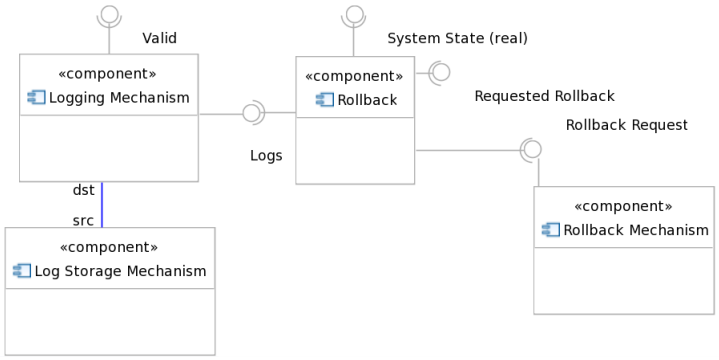

Figure 2: An UML component model of the components we will be developing

DSL elements to store in log

Feature	Necessity
Package Name	Yes
Version number	Yes
Package fully qualified name	Yes
Target package version	Yes
Binaries associated with package version	Yes (local or remote)
Configuration files of target package version	Yes (for merging)
Connection to remote repository	Depends where files are stored
Current version binaries	No (Unless the roll-back fails)
Snapshot of the entire system	No (only granularity of package)
Additional processing time	Yes
DSL logs	Yes (for our implementation)
User input	Depends (for feedback or selection mechanisms)

Table 5: Roll-back requirements

storage mechanisms investigated in Table 4 a Berkeley Database would appear to be a suitable mechanism for storing the logs. It is not fully ACID compliant but it does maintain ACID transactions [1]. SQLite is another potential database we could use for storing the log data. The ACID property that it does not enforce is that of integrity as it allows any type of data to be inserted into any field without checks. By adding an application level check this ACID property would be enforced and we could use it as an alternative to Berkeley. RPM already uses a Berkeley system so in terms of programming consistency it also makes some sense to re-use the components already established for our implementation but it is not essential and can be chosen by the vendor.

[1] http://blog.r1soft.com/2009/11/02/the-fine-print-on-file-system-journaling-%E2%80%93-part-3/

System	Database	ACID compliant	DBMS required
Berkeley	Yes	Full	No
Stasis	No	No	No
MySQL	Yes	Partial	Yes
SQLite	Yes	Almost	No
Oracle	Yes	Yes	Yes

Table 6: DSL log storage mechanisms

Berkeley Full ACID compliance [2] Other types of database [3] Description of Berkeley DB Engine [4] Isolation and serialisation [5]

[2] http://products.databasejournal.com/dbtools/ss/Sleepycat-Software-Berkeley-DB.html

[3] http://www.amaltas.org/list/database.html

[4] http://www.linuxplanet.com/linuxplanet/tutorials/6034/7/

[5] http://sheeri.com/content/isolation-%2526amp%3B-concurrency

5 System Architecture Definition

The architecture of a roll-back system is not trivial and can be achieved in many ways. Using pseudo-code to demonstrate the structure of the roll-back mechanism it may become more apparent how a DSL benefits the roll-back mechanism that will be implemented.
In Listing

```
Read TiD ⇐ UserInput
foreach STiD in TiD do
    foreach DSL_Statement in STiD do
        if DSL_Statement ! = reversible then
            Exit "Not Reversible"
        end
        else if STiD = TiD then
            Reversible = "true"
        end
    end
end
```

Algorithm 1: Algorithm to detect if all the children are reversible

Once we have detected whether or not all the sub-transactions and the transaction itself is invertible we then need to proceed with the roll-back.

```
foreach STiD in TiD do
    switch type_of_transaction(STiD) do
        case update
        end
        case remove
        end
        case downgrade
        end
        case install
        end
    end
end
```

Algorithm 2: Second sub-algorithm

Where

- TiD = Transaction ID
- STiD = Sub-Transaction ID

```
InverseDSL(Pkg_V)
foreach DSL_Statement in STiD do
    switch DSL_Statement_Type do
        case DSL_Invertible
            if FilesModified then
                StoreFilesInFlatFileSystem()
                CreateLinkInBDB()
                StoreLogEntry()
            end
            else if ConfigurationModified then
                StoreCurrentSettingsInVCS()
                LoadPreviousConfig()
                if UserChoice then
                    RequestInput()
                end
                else
                    Either use old files or try to merge
                end
            end
        end
        case VCS
            CheckCurrentState()
            DetectFilesPreviousState(TiD)
            ReplaceFiles()
            SaveStateFlag(true)
        end
        case FS_State
        end
    end
end
```

Algorithm 3: Third sub-algorithm

#	Requirement	Designation
M1	REQUIRED	Transaction respect ACID properties
M2	REQUIRED	Transaction Independence
M3	REQUIRED	Package level granularity
M4	REQUIRED	Monitoring domain
M5	REQUIRED	Device independence
M6	REQUIRED	Minimum disk space use
M7	REQUIRED	User transparency
M8	REQUIRED	File-system Independence
O1	MAY	Permanent change monitoring
O2	MAY	User interaction

Table 7: High level specification of transactional upgrade system with rollback capabilities.
[1]

- DSL_Statement corresponds to an individual statement in DSL.

In our architecture we must hold to the principles mentioned in Deliverable 3.1. Although a more intensive work in the scope of Mancoosi's Workpackage 3 has to be performed, a first approach to specification of a transactional upgrade system with rollback capabilities should follow the high level specification[2] in table 7. A brief description of the mandatory elements of the high level specification follows. This is a slightly modified version from that of Deliverable 3.1 where the majority of the criteria were selected.[1]

- **Respect ACID properties**: the system should respect the ACID properties: Atomicity, Consistency, Isolation and Durability.
- **Transaction independence**: One transaction should be independent of other transactions. This means that a rollback of a transaction has to be executed without needing other transactions to be rollbacked as well. This condition has to be slightly relaxed to accommodate for parent and sub-transactions. Sub-transactions must be rolled back to allow parent transactions to roll-back. Different transactions should rely on different children transactions and be independent from each other though.
- **Package level granularity**: the level of granularity of the transaction is the package. This does not block the possibility of maintaining state of files independently and allow some operations to be performed at that level. We will try and maintain the state of files but in terms of roll-back the target will be a package version, not the state of a file.
- **Monitoring domain**: A rollback system should monitor all files related with

the package. This files comprise not only the files included in the package but all files created, modified or deleted during the transaction.

- **Device independence**: the rollback capability should be available in different devices like servers, desktops, mobile devices and others.
- **Minimum disk space use**: the used disk space for storing transactional information should be limited to the absolute necessary. The disk space should be limited to 5% to 10% of the total disk space used by the system. This property is not the most important for roll-backs that are performed infrequently but as the roll-back system will be on all the time it is important to make sure that we are storing information efficiently.
- **User transparency**: If activated, the support for rollback transactions should not oblige the user to specific actions or decisions. Automatic roll-back mechanisms should be 'safe' and should maintain information at the cost of storage space and efficiency. As with traditional meta-installers, current user choices should be maintained. [1]

The following element is crucial for the uptake of any proposed system:

- **File system Independence**: a transactional upgrade system must be independent of a specific file system. Requiring a certain system state or file-system means that porting a transactional update scheme would require extra changes to incorporate the differences in the system types. An ideal system will be file-system and distribution agnostic.

The following elements may be implemented but are not mandatory:

- **Permanent change monitoring**: permanently monitoring the supervised files without being triggered by the user or the transactional system may be an option of implementation. This would involve a monitoring daemon and catching any changes to files or system variables. Aside from the details of getting such a system to work, to be able to store the large amount of information generated it would be necessary to realise when a change has occurred and to save the new information. This is similar to an automatic, scheduled snapshot system.
- **User interaction**: the transactional system may allow the user to add new files to be supervised. It may be that the end user would like to store more than just the system state and perhaps important files. However as our system focuses on package management we would just be storing the files in a VCS or database and this would be like widening the scope of a snapshot system. It is certainly possible but is not the core focus of using a DSL supported mechanism. [1]

Order of preference	Type of roll-back
1	1-1 mapped inverse DSL command (DSL)
2	From the DSL of the forward command try and create an inverse (DSL)
3	Analysis of configuration states before and after (DSL)
4	Perform the roll-back using the maintainer script files as before
5	Do not perform the roll-back at all and inform the user.

Table 8: Order of precedence of inverse DSL selection

If this mechanism fails then there may be other facilities to try and attempt the rollback but currently we will define that the roll-back using the DSL directly is not possible. We can then perform the remove scripts associated with the package and analyse which reverse mappings it performed. If a roll-back is not possible then we will provide the user with feedback explaining the situation. Notes: If ParentID and DSLiD are the same then the DSL command is the head of a transaction. Null would refer to an orphaned value. Are parentID and transactionID equivalent terms or can one be derived from the other? Another issue is in terms of granularity of capturing changes as identified before, on two (using scope generally) axes; time and scale. The overall size of our transactional logs will be determined by the scopes of resolution and how often we capture changes. If we frequently capture all the changes of all files on a system then we will very quickly run out of space on the system and there will be frequent usage of the capturing system we use. Less frequent and less focused scopes of resolution will possibly not capture all the necessary information to perform roll-backs. The overall mechanism can be seen in terms similar to that of the Nyquist Theorem for capturing audio. We need a sufficient capture rate and sufficient information stored to roll-back. Unlike in backup-mechanisms and snapshots it is important to maintain all previous changes as we require the sequence of steps to resolve back to a previous point. A backup point using the proposed system would involve collating all the saved system model states and the DSL transaction logs.
This process is fine for elements that have a direct reverse command and that the command is bundled with the rest of the DSL code in the packages. Whenever such a modified package is downloaded it has the corresponding DSL entries and we can store both into the database when they are encountered. This assumes that:

1. we will be modifying packages to have an additional set of DSL commands.
2. we are able or have the DSL inverse commands available.
3. the reverse commands provided by the package maintainers are correct.

Point 1 is quite a large issue to solve non-programmatically and it also adds additional overhead to the size of packages. By the definition of the DSL being expressive enough the DSL should be able to replace the current maintainer scripts

entirely but for backwards compatibility with non-DSL compliant installers this will have to be left in. We are then left with the possibility of back-porting a selection of packages to have DSL compliance. We can either then store the DSL commands within the packages themselves as an additional element or using a unique key, describing the package we could link to the corresponding DSL file. This is similar to the system employed by Conary where packages link to a maintainer file on the rpath webserver that link packages to changeset files. Another option is analysing the maintainer script files in-situ and interpreting them into a set of DSL statements.

Location	Size difference	Local	Remote	Backporting	Unique Key
RPackage i)	-config +DSL size	No	Yes	Yes	No
APackage ii)	+DSL	No	Yes	Yes	No
Ext Package	DSL	Possible	Possible	No	Yes
In-situ	Temporary (+DSL)	Possible	Possible	No	No

Table 9: Locations to store the DSL

Point 3 can be addressed by using the failure detector at a later stage to see if the configuration states before and after the roll-back are consistent regarding the system meta-meta-models. If there is a failure we can then perform an immediate roll-back or inform the user of the mis-matched configuration states.
Point 2 however is more complicated to resolve. If there are no direct DSL invertible commands or the commands require a capture of the state of the system we then need to use additional facilities to perform the roll-back.
There are many types of files associated with packages that we will maintain in our system and depending on the type of file it may be better to maintain a local copy, a server-side copy, maintain it in a Version Control System (VCS) or a database. To promote discussion there is a table of files commonly found in packages, Table 5. Some of these files are candidates to be stored using particular mechanisms because of the type of file and their role. The main issue is that binary files are difficult to maintain in terms of differences in versions. A small change in a binary file, e.g. changing a variable is likely to be difficult to track down as the whole file will have a different checksum but it is difficult to isolate the change. Media files are one example of binary files that do not change often between packages usually and are interesting to identify as they tend to be large files even when compressed in archives and may not change that often. If we are to maintain every revision of a package in a repository it will be of importance to minimise the usage of redundant files and save space. Links to previous versions of the file that have not been modified could be stored in terms of sym-links in the repository. This is a server-side issue but it means that for any package we should be able to point to a resource and the server should resolve it to that of a new file or that from an old package. Also it may be that package maintainers may choose to re-arrange the layout of the package. Documentation for instance may be consolidated into a directory ./docs/ rather than at various locations previously by a maintainer. If this

is the case files may not have changed but the package layout will have changed. Using the DSL and sym-links on the server side it may be possible to reduce the size of large repositories by identifying redundant files. Dynamic package creation using checksums of files and DSL is not the main consideration but it is something to look at given that roll-back will need access to old revisions of files and is not currently performed.

Executable files are usually, unique to packages and when a new version of a package is released it is normally the executable code and formal logic that is modified. As they are normally not used by other packages they can generally be stored as different versions and as they are binary a VCS would not be suitable to store them. Instead the executables should be stored as files on a per-package basis.

Documentation files are an example of an ASCII file that could be maintained in a VCS locally. The changes are likely to be additive between revisions with modifications of version numbers and reworked text. If any changes are merged incorrectly they are unlikely to cause a package failure and so can be merged with less safe-guards. They also generally are not modified by the end-user.

Configuration files on the other hand are modified by the end-user and have to be kept in a working state. Particular elements may be added or removed by an upgrade and others will be done by the end-user. These changes have to be merged together to keep the configuration consistent. For the example we will compare the cupsd.conf file between instances of CUPS version 1.2.0 and 1.4rc1 as shown by Table 5. There are many entries that are identical, some which are modified, some removed and others which are completely new entries. Removing and adding entries are the easiest to deal with as they do not need resolving. Modified elements and elements that are modified by the end user after changes have been made are more complicated. For instance if some modifications are made to the elements by an end user these changes may or may not necessarily be backwards compatible. If a reference is made to a new feature that was not available in the previous version of the package should that value be maintained, maintained but hidden or completely removed. These are just a few of the possibilities available when merging. It might not always be possible to know what sort of merging should take place though.

Version 1.2.0	Version 1.4rc1	DiffType
# Log general information in error_log - change “info” to “debug” for	# Log general information in error_log - change “@CUPS_LOG_LEVEL@” to “debug”	Modify
	@CUPS_SYSTEM_AUTHKEY@	Addition
Allow localhost		Removal
# Sample configuration file for the Common UNIX Printing System (CUPS)	# Sample configuration file for the Common UNIX Printing System (CUPS)	No change

Table 10: Some snippets of configuration differences between versions of CUPS

File-Type	Example	ASCII/Binary	Diff	Local	Server
Executable	cupsd	Binary	No	No	Yes
Library	libcups.so.2	Binary	No	Yes	Yes
Documentation	README.txt	ASCII	Yes	No	Yes
Media	cups.png	Binary	No	No	Yes
Configuration	cupsd.conf	ASCII	Yes	Yes	Yes
Resource files	cups.desktop	ASCII	Yes	No	Yes
Archive	laserjet.ppd.gz	Binary	No	No	Yes
Template	users.tmpl	ASCII	Yes	No	Yes

Table 11: Different types of file to monitor

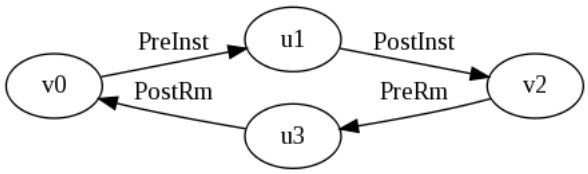

Figure 3: The evolution of a package using configuration scripts

6 Roll-back transaction definition

If a pure DSL command or set of commands does not perform the roll-back sufficiently we will need to use additional information that the DSL has provided in combination with other facilities to provide roll-back. The System meta-meta-model file will maintain certain aspects of the state of the system. If provided with the state of the system before and after an operation takes place we can view the change in states and try to ascertain which were as a result of the DSL.
We can capture all the elements that change during the configuration by trapping all system calls. A file caught by this kind of trapping might be similar to the example seen in Table 6. However there are many other details that could be captured and this would be implementation specific. Also there would be many files that have nothing to do with the package in question that get modified during the two system states (before and after) and would possibly get captured. This blanket coverage of the file-system/monitored files is akin to taking snapshots of the system before and after a change has taken place. Of course there are the problems associated with large snapshots such as how often to take the snapshots, where to store them, garbage collection, snapshot window to data change rate, granularity amongst others. With the DSL we are capturing all commands that are forward acting but the problem is that we might not have the reverse DSL commands available. We therefore have a large hint in terms of what we expect to be modified. We can therefore analyse the DSL statements and narrow the focus of what we will be capturing with file-system captures or any other methods to the meta-classes of the system that we expect to be modified. For instance if we perform an action that we know will be modifying a set of files in a folder `~/home/.arb_data/` but don't know exactly how we can then combine the other mechanisms that we will discuss and this one to capture the state of the folder before and after. Again if we know that certain environmental variables are likely modified but we have know exact knowledge of how they will be modified we can capture certain environmental variables that are modelled within the system and then again afterwards and notice changes and either deduce what happened or just capture the change. With file manipulations we for instance don't expect environment variables to change and so can capture just file changes for the associated DSL statements. It might be that the DSL commands that we produce from the before and after case are insufficient to capture what is changing in which case we will have to know when we fall back to using the standard maintainer script files and when to abort the transaction. It has already been found in research by Olivier Rosello of Mandriva for MANCOOSI that there are subtleties to do with a wide range of issues such as implicit library dependencies amongst others. As it may take time to be able to resolve these issues our architecture has to be designed such that if a package is not a candidate for roll-back that it will inform the user and perform at least as well as the current breed of package managers. From an implementation perspective what this means is that we will have to allow the roll-back mechanism to pass control back to the meta-installer if we are unable to come up with a suitable strategy for roll-back.

Files before	chksum	TransactionID	Files after	chksum
/etc/cups/classes.conf	8838...	1	/etc/cups/classes.conf	758b...

Table 12: The file system before and after a transaction

One of the issues of the implementation is knowing whether or not a reverse DSL command exists. If a pure DSL statement can be recorded that would alter the system to the state before another DSL statement then it is the logical inverse of that DSL statement. Knowing if we can roll-back or not means that we can look at a transaction and determine if a roll-back can be performed on it. For instance take the cups example we have been using and a set of transaction and DSL statements that are recorded as a result in Figure 4. What we can say is that a transaction subtree is suitable for roll-back iff all the sub-elements have roll-back DSL equivalents. If any of the sub-elements of a transaction cannot be reverted then there is no chance of performing a full roll-back of the parent transaction. If this is the case then we can quickly determine if a roll-back should be possible for a large set of sub-transactions by looking at the leaf elements and seeing if any of them do not have associated roll-back elements. It may still be possible for a roll-back of the system even if sub-elements do not have DSL syntax.

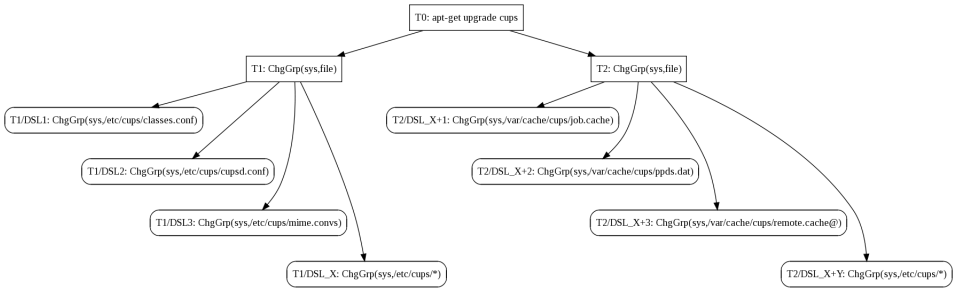

Figure 4: TransactionID Based Tree

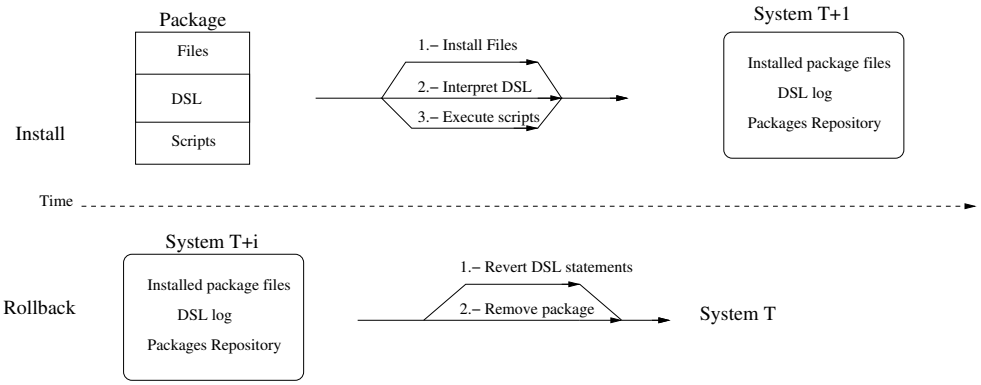

Figure 5: Install and rollback process

7 Example of package roll-back

In the following section a specific example of a package will have its maintenance script files converted into DSL statements and captured accordingly. By working through this example it may simplify the process that is envisaged for the implementation mechanism. For this example we will use the architecture defined before and the methodology of Deliverable 3.2 to describe what will happen on a system that has a DSL supported roll-back mechanism.
The Package Meta-Installer that will be modifyied, apt-rpm into DSL_TTSG_RPM runs off rpmLib = 4.4.6. The module which has been identified as being the one to modify is that of rpmcc.c.
Listing 1 refers to the post install configuration that is run by the Meta-Installer after installing CUPS.

Listing 1: Maintainer script file

```
cups.spec.post
%post
# Make sure group ownerships are correct
chgrp -R sys %{_sysconfdir}/cups %{_var}/*/cups
```

The files found in Listing 2 are representative of those which would be modified by the maintainer script. We capture as a list all the DSL statements modifying the individual files. Using the log we can then see which DSL statements are recorded in a transaction and if directly available run the inverse DSL commands or use the parameters to see what was modified and by which DSL statement. For simple cases such as this where the files will have their permissions changed we can capture what they changed from and to and the command is easily invertible but not self-inverting.

Listing 2: Sample file structure

```
/etc/cups/classes.conf
/etc/cups/cupsd.conf
/etc/cups/mime.convs
/etc/cups/printers.conf
/etc/cups/snmp.conf

/var/cache/cups/job.cache
/var/cache/cups/ppds.dat
/var/cache/cups/remote.cache
/var/cache/cups/rss/

/var/log/cups/access_log
/var/log/cups/error_log
/var/log/cups/page_log

/var/run/cups/cups.sock
/var/run/cups/certs/

/var/spool/cups/
```

Listing 3 collapses the expanded tree of all the files that will be modified as per Listing 2. This can be expanded by an implementation that looks recursively through

the directories and is less redundant. However for the initial implementation it is better to explicitly log all the information serially and think about compressing the information at a later stage.

Listing 3: Sample directory structure

```
/etc/cups/*
/var/cache/cups/*
/var/log/cups/*
/var/run/cups/*
/var/spool/cups/*
```

Listing 4: Equivalent DSL Grammar

```
SCRIPT :: STATEMENT_LIST :: {CONTROL_STATEMENT :: {CASEPOSTINST ::
    CasePostinst(){CONFIGURE :: configure :: STATEMENT_LIST ::
    ITERATOR_STATEMENT :: ITERATOR_DIRECTORY}, STATEMENT_LIST :: {
    TEMPLATE_STATEMENT :: {CHGGRP}, \varepsilon}, \epsilon }ABORTUPGRADE
    :: \epsilon ABORTREMOVE :: \epsilon ABORTDECONFIGURE :: \epsilon },
    STATEMENT_LIST :: \epsilon
```

DSL Grammar in a diagrammatic tree

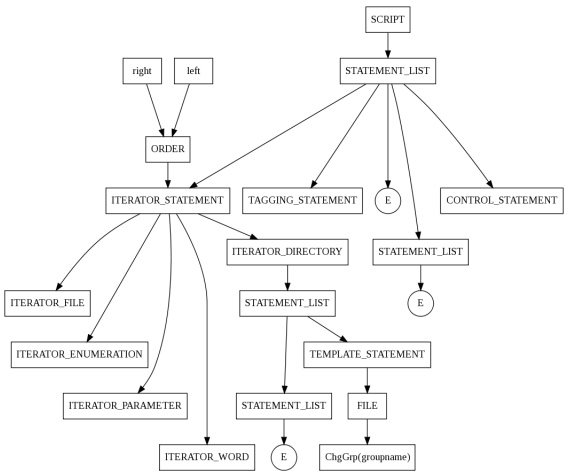

Figure 6: A tree showing the sub-section of the grammar tree related to CUPS post inst script

Listing 5 is an example of how the CUPS maintainer script would be represented in the ATLAS Transformation Language (ATL). This representation allows the model of the system developed as part of Work Package 2 to be transformed from a source to a target configuration. The exact semantics of the rules will not be exactly as per this listing but it is demonstrative of how the simulator and failure detector will modify their internal properties based on DSL commands. ATL commands onto the model are not commutative and as such the order in which they are performed is important. They will be run in the same order as the DSL statements in the log mechanism.

Listing 5: ChangeGroupRule

```
rule file_chg_group(file_name,group_name){
using{
Environment!SystemDirs
}
do{
file_name.Groupname <- group_name
}
}
```

Listing 6: Equivalent DSL Commands

```
ChgGrp(sys,/etc/cups/classes.conf)
ChgGrp(sys,/etc/cups/cupsd.conf)
ChgGrp(sys,/etc/cups/mime.convs)
...
...
```

Listing 7: Inverse DSL Commands

```
ChgGrp(owner1,/etc/cups/classes.conf)
ChgGrp(owner2,/etc/cups/cupsd.conf)
ChgGrp(owner1,/etc/cups/mime.convs)
...
...
```

Now if we want to reverse the set of DSL statements performed by:

```
chgrp -R sys %{_sysconfdir}/cups %{_var}/*/cups
```

we could perform a number of different procedures. The simplest is iff we store the invertible DSL statement along with the DSL command, we can just find the associated Transaction ID, check to see that all elements are invertible, and run the inverse DSL statements. If however a direct mapping does not exist there may exist a method by which we can capture the original configuration and through a capture of the system state afterwards be able to compare the differences and create a reverse mapping in terms of multiple DSL statements.

8 Implementation

8.1 Overview

All of the previous sections have mentioned the areas that we will have to explore and work around to get a working implementation of transactional roll-back using the DSL. There are a few named techniques as well as other techniques that have not been investigated that all try and cover the problem of rolling back software updates and restoring previous system configurations. What is clear is that the amount of research that has gone into investigating roll-back, falls far short of the research into upgrading software systems. This is understandable as after all software developers try to release software that is functional and that each successive iteration improves upon the previous version of the software. Rolling back software can therefore be seen as counter-productive. There are times though that the end-user or system administrators would want to be able to roll-back to a previous configuration. The options available for roll-back have been identified as insufficient and as such could benefit from the research carried out as part of the MANCOOSI project. As part of Work Package 2 using a Domain Specific Language (DSL) was decided upon as a potential way of solving some of the problems associated with the granularity of package management and to provide a new framework from which package maintainers and users can benefit from this new technique.

8.2 Timeline

A proposed timeline has been suggested for discussion with @rpm5.org and the other members of MANCOOSI as well as serving as an internal document that can be used for reference. It might be possible that the timeline is unworkable or that there are delays or that it needs reconsideration in which case it will serve as a base from which we can refine milestones and co-ordinate. The main deadlines are that of the March review for the MANCOOSI project and also that of the FOSDEM 2010 conference of 6^{th} and 7^{th} February where we are looking to present some of our findings. We are aiming to have a working implementation for FOSDEM and to refine it in time for the March Review of the project. @rpm5.org timeline is only for reference as other commitments may interfere with the development process but generally the goals and milestones identified will be held to.

8.3 Logical Blocks

There are many areas that need development for working with the results of Work Packages 2 and 4. The key areas that have been identified as sections that need to be addressed as part of Work Package 3 are:

- Roll-back mechanism
- Log storage mechanism

MANCOOSI
Caixa Magica Software

Project Coordinator: John Thomson
Today's Date: 1/4/2010 (Mon) (vertical red line)
Start Date: 11/2/2009 (Mon)
First Day of Week (Sun=1): 2

WBS	Tasks	Task Lead	Start	End	Duration (Days)	% Complete	Working Days	Days Complete	Days Remaining
1	**Organisation & Planning**	John	02/11/09	21/12/09	49	29%	36	14	35
1.1	Creation of time-sheets and logistics		02/11/09	16/11/09	14	50%	11	7	7
1.2	Resource Allocation		09/11/09	16/11/09	7	0%	6	0	7
1.3	Review of Implementation	Paulo	17/11/09	19/11/09	2	0%	3	0	2
1.4	Mid-task Review	Paulo	20/12/09	21/12/09	1	0%	1	0	1
1.5	Final Review	Paulo	27/1/10	30/1/10	3	0%	3	0	3
2	**Caixa Magica Software Inmplementation**	John	02/11/09	06/2/10	95	1%	70	1	94
2.1	Design/Specification of Implementation		02/11/09	16/11/09	14	15%	11	2	12
2.2	Programming		20/11/09	21/1/10	62	0%	45	0	62
2.2	Investigation		10/11/09	24/11/09	14	0%	11	0	14
2.5	Database design and impl.		24/11/09	15/12/09	21	0%	16	0	21
2.4	TPPM mechanism		15/12/09	30/1/10	46	0%	34	0	46
2.3	Simulator/Failure Detector Integration		09/1/10	30/1/10	21	0%	15	0	21
2.2	Contingency		29/1/10	05/2/10	7	0%	6	0	7
2.6	Final Test of Implementation		22/1/10	05/2/10	14	0%	11	0	14
3	**Jeff Johnson @rpm5 Implementation**	Jeff	02/11/09	05/2/10	95	18%	70	17	78
3.1	Design/Specification of Implementation		02/11/09	16/11/09	14	90%	11	12	2
3.2	Programming		02/11/09	21/1/10	80	50%	59	40	40
3.3	RPMDB Schema		02/11/09	13/11/09	11	50%	10	5	6
3.4	Conversion Tools		13/11/09	13/12/09	30	0%	21	0	30
3.5	Unit Tests		13/11/09	23/12/09	40	0%	29	0	40
3.6	Logging representation		20/11/09	20/12/09	30	0%	21	0	30
3.8	Attaching Secondary Stores		29/11/09	05/1/10	37	0%	27	0	37
3.8	Hearts and Minds		02/11/09	31/1/10	90	5%	65	4	86
3.9	Final Test of Implementation		22/1/10	05/2/10	14	0%	11	0	14
4	**Events**	N/A	02/11/09	02/11/09					
4.1	Christmas/Natal		21/12/09	04/1/10	14	0%	11	0	14
4.2	Thanksgiving		25/11/09	28/11/09	3	0%	3	0	3
4.3	Nice MANCOOSI meeting		06/1/10	09/1/10	3	0%	3	0	3
4.4			06/2/10	06/2/10	0	0%	0	0	0
4.5	FOSDEM		06/2/10	09/2/10	3	0%	2	0	3
4.6	March Review		24/3/10	26/3/10	2	0%	3	0	2

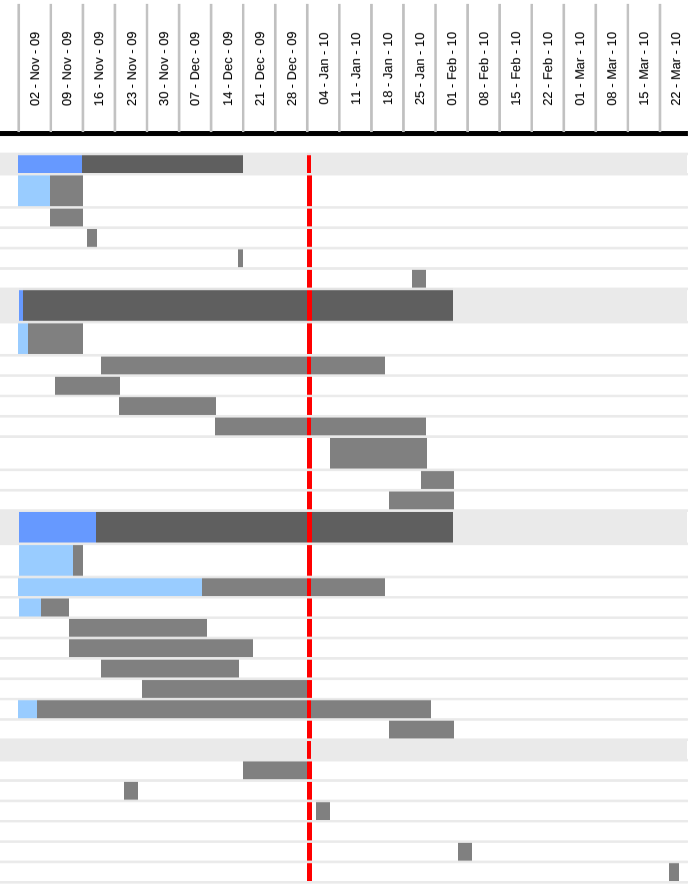

Figure 7: Proposed Timeline

- File and configuration storage mechanism
- Server communications and interfacing
- Integration with rpmpm.cc
- Communication with Failure Detector
- Interfacing with Simulator

The log storage mechanism is one of the most important areas of the overall roll-back system. Having a serialised and consistent log is essential as we will rely on the information stored in the log to move between configuration states. Any shortcomings or corruption within the log will mean that we are not guaranteed to reach a valid configuration state which is usable with the simulator and failure detector. It is therefore essential if we are going to maintain a representation of the system in terms of a Domain Specific Language that we store the representation using mechanisms that guarantee the data are in a known state. For this implementation branch we will be using SQLite3 to store the logs.
The first main implementation point is how much we will use the DSL for the package installation process. For this first version of the approach we look at using the DSL as a log fabricator that will store the effects of DSL as specified in a .dsl file included in some way with the package. These logged DSL statements will provide the first approach for tackling roll-back. Also in this implementation scheme we look at providing hooks to a simulator and failure detector that will be created as part of the work in Work Package 2. The log mechanism will store all the DSL commands that change a configuration A to A'. By capturing these changes we hope to be able to revert back from configuration A' to A. The reasons for doing this are mentioned in D3.2 and in the supplementary document, Rollback Component Definition.
Log file As mentioned the log file will capture the DSL commands and the consistency of such commands are integral to having a reliable mechanism from which we can move between package configuration states. If we cannot be assured of the state of the log or the data contained within then we cannot guarantee that we are in a given state. For this reason preliminary work and research has been carried out into how we will store the logs. Two main contenders arose, Berkeley DB and SQLlite3. There are other options available such as Stasis and using fully fledged database management systems. The choice has been taken to use SQLlite 3 due to familiarity with SQL of the developers involved. Berkeley DB has been investigated but as JJ has already implemented such a mechanism and that time constraints are now the major issue it is seen as a suitable tradeoff between complexity of a first implementation and the resulting efficiency benefits.
As for the roll-back mechanism we defined at the beginning of the document that any mechanism chosen would be selected in such a way as to minimise the cases where roll-back is infeasible. The choices will therefore be made to maximise the potential for roll-back.

To store files in such a way that it is not file-system dependent means that files should be stored either remotely or in cases which are not suitable for remote storage, to use a storage system that is independent of architecture and system used. There are arguments both ways for whether to store binary data in databases or to have the data stored in a file-system. As for our new proposed implementation we are trying to keep it as system agnostic as possible and hence will store files in the file-system and link to them with a database. As we are already proposing to use BerkeleyDB we can also store locations of files in this way. If at a later point we decide to compress the files or do something else with them, they will not be contained within a single type of database but rather be associated with a location. This does mean that we lose some of the safety mechanisms of storing files within a database and there is the potential that the files could be modified by the end user or become corrupted but there are mechanisms such as digests that we can use to counter-act this.
In terms of communication with the repository servers we will use the mechanisms already offered in apt-rpm, namely rpmlib, to name a specific version of a package that we intend to use. Creating a new system for communication with the server is not necessary but the mechanism by which the servers store the information has to be addressed. If we intend to store all the binaries for all versions of the packages on the repository server then we will need to have a more efficient diff mechanism to maintain the changes between packages. A VCS mechanism that maintains just the difference in the source that generates the binary files would be ideal but as we are using the binaries, VCS tend to be less efficient for storing data. There exists a few solutions for storing binary files in a VCS manner. One such solution is Checkpoint [6]. Bsdiff [7] can be used to compare binary files and to create small patch files. Courgette [8] is a system that is being developed by Google as part of the Chromium OS and allows even more efficient storing of changing application files. Sending small updates from the server will mean that the whole package will not have to be resent but rather the elements that have changed. Initially though the files to be restored can be stored locally, then moved to a remote server and eventually any mechanism that reduces the amount of redundant information stored and transmitted between versions of software can be investigated at a later stage.
As for integration with the existing apt-rpm and more specifically having a version that implements the usage of the DSL we are going to use a new fork in the software branch. As stated before, apt-rpm currently uses rpmlib version 4.4.6, the same as that of Yum. Within the facilities provided by rpm we could choose to maintain functionality or create new code to realise our aims. The dependency solving, conflict catching facilities available in rpmlib as well as the facility to create package transactions has been investigated and it would appear a suitable location for inserting our new code. Of the sections of rpmlib, the install packages function and

[6] http://code.google.com/p/checkpoint/
[7] http://www.daemonology.net/bsdiff/
[8] http://www.chromium.org/developers/design-documents/software-updates-courgette

more specifically just after the topological sort on the transaction seems to be a good location to insert our software. From the list of packages to be installed we can then take any information about those packages and run a simulation of what installing the packages would do to the state of the system. At this simulation stage we could decide if a roll-back was possible or not and decide whether to continue with the installation of the system, either automatically or with the assistance of the user.
The communication with the failure detector by design will be quite limited. If we have received an error from the failure detector it is likely that it will have handled the communication to the end user so what is left is to decide on and implement a roll-back plan. At this stage we will need error checking to make sure we are not in an infinite loop searching for a plan that is infeasible. Once the error has been rolled-back we then need to inform the end user of the current state of the system and what has happened.
Communications from the simulator will be more complicated as the roll-back mechanism will suggest a strategy for roll-back and if we wish to simulate this we will have to interact with both the input and output of the simulator.

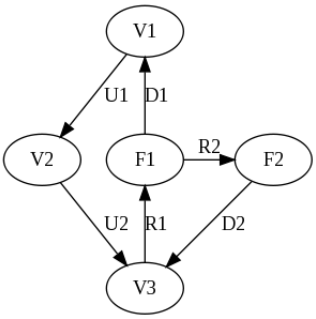

Figure 8: State Transition Diagram of package versions

Get the TiD corresponding to the package version that we want to roll-back to. Check the current file digest with that of the files in the target system. If they have been modified we should preserve the files we currently have that will be modified as part of this system-state. Depending on the files changed we use different mechanisms to store the changed information. Configuration files for instance will be stored in a VCS, whereas binary files may use the Courgette system and have a patch applied to them to restore them to the target state. These patches would need to be downloaded from the repository server or created by using a combination of the DSL and other mechanisms that will be explained in more detail. The configuration files will be stored in a VCS as the current configuration. We can analyse the DSL elements and suggest which files might be configuration files through the analysis of the actions performed on them. Then need to request from the server the old versions of the binaries or the patch files. Only need to download files for which the files have changed and so we can reduce the amount of

information needed to be stored and transmitted by the server. Could detect which files are modified using a combination of pstrace and other file monitoring mechanisms. We could also generate packages on the fly using the information from various sources and then send the client the requested package or differential in the package. In the first instance a simple mechanism would be to retain all the old binaries rather than patch back to the old version. Next we must run the pre-removal DSL equivalent scripts to make sure that any services that need to be de-registered can be performed etc. Applying the patches and returning the file-system to the state of the previous stored configuration is the next step and uses the files we have downloaded. Either apply patches or overwrite the target files with those of the original system. As for the configuration files we will then request to the end-user if a verbose mode is selected, whether or not they want to roll-back the configuration files. The choices will be to keep the configuration as is/use the configuration stored on the server/use the configuration stored in the VCS before the update was performed/ or lastly to merge the changes. If no user input is used, then to maintain the maximum potency of the roll-back mechanism we will use the stored configuration file from the VCS. We can then run the failure detector and see if there is a configuration mis-match and if so advise the user if a roll-back of the roll-back is required. The implementation would be the same as the roll-back used here but that the target is the state before the original roll-back and the source is the failed state that we are in. The effect would be akin to that of reverting a snapshot. Roll-back of a roll-back may yet return as a failure but if the configuration is the same as that before the roll-back was performed then we have got back to the original system state and will stop at this point.

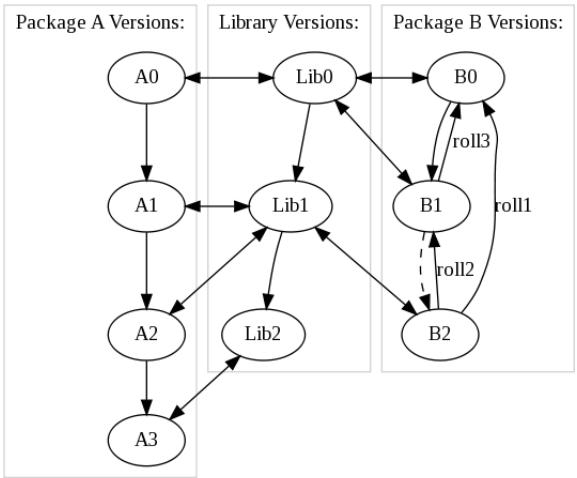

Figure 9: State Transition Diagram of packages dependent on libraries

One of the problems when rolling back that has been identified by Olivier in his research for Mandriva's implementation strategy is that of library dependencies.

Figure 9 helps to indicate some of these problems. It can be further expanded when considering all the possible roll-back states for these two example packages as shown in Figure 10. Returning to Figure 9 if the roll-back arrows that are indicated by the arrows roll[1-3] are to be successful then a few things have to be identified. Rolling back Package B from version 1 to 0 i.e. roll3 in the Figure is the simplest scenario. Both version 1 and 0 of Package B are dependent on and use Lib0. Rolling back the Package B therefore has no implicit effect on Package A through the modification of Lib0. What is more complicated are the roll-backs indicated by arrows roll2 and roll1 in Figure 9. If Package B is to be rolled back from version 2 to either 1 or 0 there are now additional considerations. If Package B versions 1 and 0 are only dependent on having a library version greater than 0 then again we can be reasonably assured that rolling back Package B to either of the versions will lead to a valid state. What is more complicated is if version 0 of B was dependent on version 0 of the Library exactly and cannot use any later editions. If package A versions, 1,2 or 3 are installed then there is a difficulty in rolling-back B in this case as rolling back the version of the Library will break those packages. If concurrent versions of the library can exist simultaneously on the system we could then re-install if not already present Library version 0 and roll-back B, leaving A associated with Library version 1. If however the versions of the library are not mutually exclusive and cannot be run on the system at the same time we will either have to roll-back all packages that are dependent on Library Version 1 to allow Library version 0 to be installed, to create a sandbox environment for the library to run in, implement some other mechanism for running two versions of the library to run or to inform the user that a roll-back is not possible. There may also exist the situation where roll-back 2 is possible and 3 and 1 are not. If this is the case we have to decide whether when the user chooses roll-back 1 whether we investigate the other roll-back possibilities and inform the user whether or not an alternative roll-back is possible. This could further be expanded to include searching for roll-backk strategies for versions older than the requested one. To maximise the number of cases where roll-back is feasible it would could be argued that searching the other roll-back strategies is important. One contrary view is that we want to maximise the number of cases where the user and only the user specified roll-back is achievable in which case we would not investigate the other possibilities. Moving to the more complicated as shown in Figure 10, we have yet more cases where roll-back stages might occur, but using the principles suggested above it should be possible to move between different versions of packages or suggest when such a roll-back strategy is infeasible. It might be necessary for instance to roll-back a collection of packages to enable a roll-back in which case it is similar to the upgrade scenario where an upgrade strategy is decided. For certain other compiled and binary resources it may not be so important to maintain exact versions for different packages. The consistency of files between states is important and so although this could be relaxed for certain types of files we will endeavour to repeat the process for all types of files. It also makes an implementation scheme easier as we do not have to segregate files (other than configuration files).

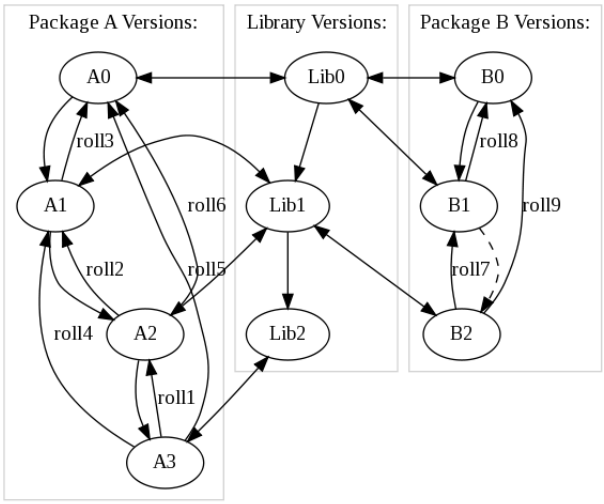

Figure 10: State Transition Diagram of packages dependent on libraries

8.4 Actual Implementation

We will be using some of the design inferences mentioned before and we will take other decisions to make a working implementation by the deadlines as stated in the Gantt chart, Figure 7. As some of the design considerations have taken longer than originally planned for some trade-offs have had to be taken to make sure a working implementation is ready in time. The design has also been simplified somewhat to make sure that it is block based and so may be more inefficient than a streamlined design but makes it simpler to refine and improve upon. The DSL will be contained in the binary package as a .dsl text file. It will be a list of DSL Commands that will be extracted and read by a utility as described later. These DSL commands will then be converted into a sequence of commands using an external pseudo-interpreter written in Python. This same Python script will then store the DSL commands into a SQLite3 DB along with any other elements dictated by the definition of the database and its schema. The main decision has been to avoid using a full interpreter that works within apt-rpm and that would have been able to parse DSL commands and perform the executions itself. Instead the original functionality of rpm will be maintained and a separate, external Python script will take care of receiving the DSL commands and storing the relevant commands to the log. Execution will for the time being, be deferred back to rpm which will use the maintainer scripts as was. The idea behind this is that we will still be able to capture the DSL elements, store them in a log and pass them to a hook that would be the simulator (that is being created as a part of WP2) and then to receive a notional valid or not valid signal and continue with the package upgrade/rollback. This is shown in Figure 12. Our mechanism breaks the normal flow just before the transaction would normally be run and control passed to RPM. For our imple-

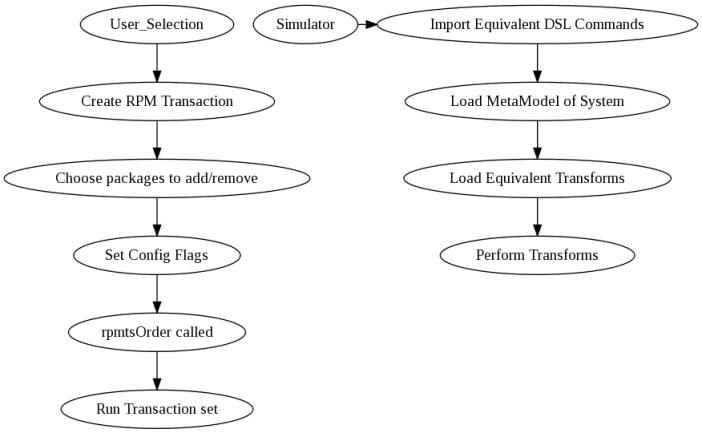

Figure 11: Inserting roll-back code

mentation, the transaction set information will be passed to the Python script that will act as a dumb logging agent. For each file it will read the DSL commands and break them down into serialised elements and store them into the SQLite3 based DSL log. Also, provision will be made in terms of call-backs to the installation scriptlets. Instead of creating a full interpreter that will be able to execute the statements what we will do is when we have picked up certain commands, link in some CLI code but for the vast majority we will initially look at passing control back to apt-rpm and let rpm unpackage, move and install the files. All we will do in the forward direction is to log what is happening but in terms of DSL. The inverse commands if they exist will be captured alongside the forward commands. To perform the inverse DSL would therefore be to run the sequence of inverse statements in reverse for the associated transaction. Linking the transactions together is an important part of the Database design as it will suggest what the limit is of grouped statements. Otherwise there is no guarantee that we will start using inverse statements from previous transactions that would lead to an erroneous configuration state.

DSL Log Schema

- DSL_ID : Primary Key Integer Unique Auto-increment
- TransactionID : Integer
- DSL_Statement : VarChar
- DSL_ParentID : Integer NOT NULL
- DSL_InvStatement : VarChar
- Dependent : Integer (boolean)

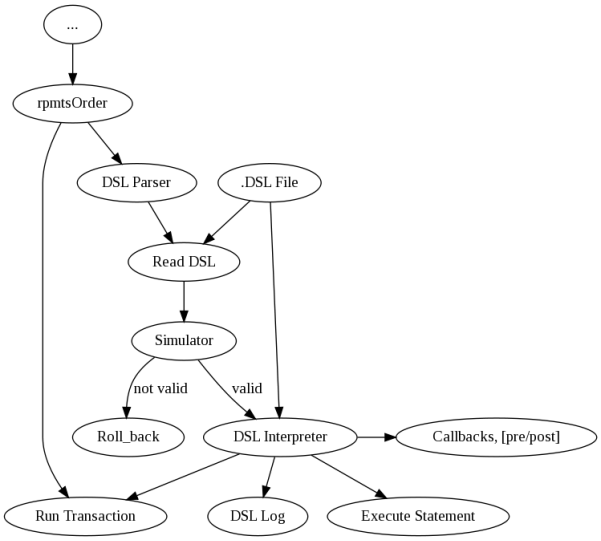

Figure 12: Roll-back implementation

The TransactionID that will be used is generated from apt-rpm, specifically rpm and encodes other details such as date installed alongside, a fully qualified NEVRA name and the changes made to packages amongst others. DSLiD will be a numeric auto-incrementing unique Key from which we will be able to explore this log. The TransactionID will however make a better index as several DSL commands will be associated with a particular TransactionID. DSLstatement is the location where we will store the serialised DSL commands that we are performing. It will take the form of DSL syntax and also will capture the variables that were passed into the command as arguments. DSLparentID is a way to group DSLiD statements together. By capturing the initial command and linking all associated DSLiD elements to a parent element we will be able to say which DSL commands relate to what. DSLInvStatement is where we will store inverse commands that are either generated from the .dsl file, manually or otherwise. This is where the mechanism will first look for invertible commands and if they are present will indicate invertibility and the commands to run, otherwise if blank it will mean that other fall-back mechanisms will be relied upon. Dependent is a place-holder in case we find that roll-back mechanism cannot roll-back without first performing a statement located else-where.

.dsl file This will be additional to control.tar.gz and .spec file scripts. Normally these files are not included in the binary files and through manipulation of apt or rpm we should be able to recover the meta data that these files contain but for convenience we will store these files in the binary archive file so we can capture information relatively easily. The .spec file information and control code is normally

encoded into meta-data that is included in the package and this could eventually be how the .dsl file information will be stored but for now it will be stored as a .dsl file in the root of the binary package as well. The terms stored will be DSL functional equivalents of the maintainer scripts that they are aimed at replacing. Rather than have to create a new interpreter solely for performing operations that rpm would already do, we are designing a system where we will use a perl/python/ruby script that will open the archive, get the associated .dsl and .spec or control.tar.gz files, serialise the commands and then save them into a log. This log will then be used to drive the roll-back. Once this has been implemented we will look at having the DSL form into an interpreter that is capable of running DSL commands and therefore eliminate the need for maintainer scripts. This however is for review once development of the other stages has occurred. The reason for a perl/python/ruby script is that they will be able to interface with BASH/POSIX shell and also to the database more simply than just a set of BASH scripts or from using pure C/C++. It may be that further down the development cycle that another development language might be preferable but the aim is to show an implementation of the DSL and as such having something that works in a block structure and is easier to segregate may not be the most efficient mechanism but for a proof of concept design it is the desired choice.

Example of what the DSL would look like in the .dsl file For CUPS therefore the DSL would be:
if :: (Environment.configuration.architecture == x86_64 , then CtrlStmt1, else CtrlStmt2)
CtrlStmt1 :: if ((FileSystem.File(/usr/lib64/cups) == File.directory) && (FileSystem.File(/usr/lib64/cups) != File.symlink), then CtrlStmt3, else CtrlStmt4);
CtrlStmt2 :: Environment.configuration.addgroup(lpadmin);
CtrlStmt3 :: if ((FileSystem.File.location(/usr/lib/cups) == File.symlink, then CtrlStmt5, else CtrlStmt6);
CtrlStmt4 :: null;
CtrlStmt5 :: FileSystem.UpdateFileSystem.File.location(/usr/lib/cups) → FileSystemStatement.iterator.directoryiterator(%File.removeFile), CtrlStmt7;
CtrlStmt6 :: FileSystem.File.location(/usr/lib64/cups).move(/usr/lib64/cuprs.rpmsave), CtrlStmt8;
CtrlStmt7 :: FileSystem.File.location(/usr/lib64/cups).move(/usr/lib/cups), null;
CtrlStmt8 :: echo (text), null; //irrelevant to DSL

//End %pre capture
CtrlStmt9 :: InstallFiles(); //Necessary to separate?
CtrlStmt10 :: InstallConfFiles();

CtrlStmt11 :: FileSystemStatement.iterator.DirectoryIterator(File.location(/usr/lib64/cups)) → (FileSystem.File.chgowner(sys));

CtrlStmt12 :: FileSystemStatement.iterator.DirectoryIterator(File.location(/var/*/cups)) → (FileSystem.File.chgowner(sys));
//Handling wild cards...
// End post capture
//REMOVAL
//Pre uninstall
/usr/share/rpm-helper/del-service

CtrlStmt13 Environment.runningServices.File.location(/usr/sbin/cupsd) → prerm_init() || → postrm_init(); //Need to think whether these still need to be split and if so need to put the postrm_init() after the removefiles() process. This DSL element removes the service from the running system akin to p.51 of D3.2. The Debian equivalent is shown on that page. For rpm systems the equivalent would be /usr/share/rpm-helper /del-service %servicename. Configuration and Package meta-class settings need to be modified. PackageSetting.services.executable.location(/usr/sbin/cupsd) → prerm_init() || → postrm_init();

CtrlStmt14 :: removefiles();

/usr/share/rpm-helper/del-group
CtrlStmt15 :: Configuration.Groups.RemoveGrp(lpadmin);

CtrlStmt 5,6,7 do not have inverses in these configuration scripts and this would? be detected by a DSL.

References

[1] Paulo Barata, Paulo Trezentos, Inês Lynce, and Davide Di Ruscio. Survey of the state of the art technologies for handling versioning, rollback and state snapshot in complex systems, September 2008. http://www.mancoosi.org/reports/d3.1.pdf.

[2] Scott O. Bradner. Key words for use in rfcs to indicate requirement levels. Internet RFC 2119, March 1997.

[3] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. Package upgrades in foss distributions: details and challenges. In HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades, pages 1–5, New York, NY, USA, 2008. ACM.

[4] Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Metamodel for describing system structure and state, January 2009. http://www.mancoosi.org/reports/d2.1.pdf.

[5] Davide Di Ruscio, John Thomson, Patrizio Pelliccione, and Alfonso Pierantonio. First version of the dsl based on the model developed in wp2, November 2009. http://www.mancoosi.org/reports/d3.2.pdf.

[6] Various. Commutativity, December 2009. http://en.wikipedia.org/wiki/Commutativity.

[7] Various. Determinism, December 2009. http://en.wikipedia.org/wiki/Deterministic.

[8] Various. One way function, December 2009. http://en.wikipedia.org/wiki/One-way_function.

[9] Various. Trap-door function, December 2009. http://en.wikipedia.org/wiki/Trapdoor_function.