Contents

1 Introduction

In this document we will describe the configuration management system RBS. RBS is not a professional package. It was written at Philips Research Labs in a machine translation project as a tool for a very interdisciplinary team to manage their software. In the course of the project it became clear that it would have been impossible to develop the huge machine translation system without RBS, although RBS was far from perfect. The most important reason for the success of RBS was, at least in my opinion, that it imposed a formal procedure on the software development trajectory. In each phase of the development it was very clear to the developer what actions he had to undertake. It should be said, however, that another reasons for its success was the fact that the people who wrote RBS were always available to provide some kind of work-around if things went wrong.
It would be a good idea to give one member of your project team the honor to be 'archive manager'. It would be the task of the archive manager to get to know all strange pecuiarities of the RBS system. This is needed to be able to take the appropriate steps if something might get wrong (which in most cases is not the result of RBS dificiencies but of irresponsible user behaviour). Together with RBS the archive manager should be able to keep the archive consistent.

1.1 Archive versus Local Workspaces

RBS distinguishes two kinds of software: local and archived. Archived software is shared by everybody in the system. This means that archived software is accessible by every user and therefore it has to be approved. The archive also keeps track of the history of all approved software to avoid the problem of untraceable modifications as much as possible. The latest version of all archived files (sources and targets) is immediately available. Local software is only accessible by those that have created it.

1.2 Consistency

One of the most important aspects of RBS is that the archive is consistent at all times. This consistency is checked by an approval procedure by which changes in the archive are checked. A major part of that procedure is to guarantee that the archive does not suffer from the problems with an uncontrolled environment and the problems with keeping all software up to date.

- all files in the archive are compiled or generated with the appropriate compiler or generator. This means that the generation of the software is completely controlled.
- all generated files are generated from other files in the correct order

- no files are missing

In future releases of RBS we will also provide a built-in procedure to enable the user to specify a test set which must be passed successfully before software may be placed into the archive.
See the section 'Mutual Exclusive Operations' for additional comments on this subject.

1.3 Releases and Components

In RBS the software has to be subdivided into releases and within releases into components. Each component in its turn should contain related software. RBS distinguishes domain dependent (DD) and domain independent (DI) components. The basic idea behind this distinction is that one might want to make a system which can be applied to several domains, e.g. a spelling checker. Such a checker may contain language independent code and language dependent code in which spelling rules are coded etc. Each domain dependent component will have its own file with spelling rules. If one sticks to the convention to give these files the same name in all DD components then it is possible in RBS to abstract in the 'make' files, i.e. in the files in which the dependencies between the different modules of the system are specified, from the name of the actual DD components.
Each component in a user's local workspace contains three sections. Each of these sections is intended to contain a special type of software. First, RBS distinguishes *genuine sources*. Genuine sources are files which cannot be generated from other files, they can be looked at as the foundation of the system that is being developed. Next, RBS distinguished *targets*, i.e. files that can be derived, generated from other files. Finally, each component has a place where they are invisible for all RBS commands, except, of course, for the command which makes them visible again.

1.4 View and Visibility

A software developer using RBS has certain software in view. This comprises all unapproved software he has made and which has not been approved yet plus the latest versions of the approved software in the archive. When a developer wants to combine his unapproved software with the software in the archive to test his local chages, RBS will use all local software plus the software in the archive, insofar as it has not been 'replaced' by the local software. In other words: when searching through the software, RBS will search among the local software first and only if it is not found there, it will look for it in the archive. It is possible, however, to tell RBS that certain files have to be ignored. Obviously, these ignored files will not be archived.

2 Installation

2.1 Archive Installation

Setting up an RBS archive takes several steps, some of which should be accepted as being just magic.

1. Log in with the user-id of the user that you want to contain the archive directories and execute the following commands:

   ```
   setenv RBS_DIR /home/rous/RBS
   set path = ( $RBS_DIR $path )
   source $RBS_DIR/setup
   rehash
   ```

2. RBS requires that all users of the same archive, i.e. the people that are allowed to integrate software, belong to the same (UNIX) group. So, before you set up the archive structure, you have to decide which group id that will be. You can choose an existing one or you can ask the system manager to create a new one. Before you continue you have to select this new group id:

   ```
   newgrp <group id>
   ```

3. Setup an archive directory structure by executing the command

   ```
   install_archive
   ```

   Now, several questions will be asked to the user:

   - RBS will ask for the path name of the archive directory to be created, e.g.

     ```
     /home/user/archive
     ```

   - Next, RBS will ask for a relative path to your release directory, e.g.

     ```
     /release3
     ```

   - RBS will ask for a list of domain dependent components. If such a division makes no sense for your application, you should specify a dummy componentname here, anyway. In the example of the spelling checker one could create a domain dependent component for each language, e.g.

     ```
     dutch english spanish
     ```

In RBS these domain dependent components have a special status, as is explained in the introduction. The RBS manager should specify at least one component here (even if the distinction domain dependent - independent does not apply here).

- Finally, RBS will ask for the domain independent components (again in list representation). E.g in the wordprocessing example one could define the components

  ```
  general utilities
  ```

The result of these actions is that a directory structure has been created. There are two components which have been created by RBS itself: `doc` and `actions`. Furthermore, a number of files have been created in the archive directory and some of its subdirectories:

`fuid`	an executable needed by RBS to change the owner of a file which is stored in the archive.
`<release>/RBS_install`	a script which is to be used by RBS for the installation of a new RBS user.
`<release>/RBS_domains`	the list with domain specific components.
`<release>/RBS_users`	a list of RBS users of the current archive.
`<release>/RBS_logicals`	a file containing the basic path definitions.
`<release>/history.rbs`	contains the history of the integration actions, i.e. who changed the archive and when was it changed.
`<release>/doc/doc_survey.rbs`	contains the documents which have been created by the RBS users, i.e. doc. nr.: user : date: filename: title
`<release>/doc/doc.make`	basic make file for LaTex documents.

4. In order to guarantee the correctness of the archive it is necessary that integration actions do not overlap. For that purpose RBS requires that they are queued. So, go to your system manager and ask him, after you have selected a machine on which you want the integration actions to be executed, to add a queue on that machine. He probably will look quite puzzled, so help him by saying : "The name of my queue is 'r', and you have to add the following line to the file `/var/spool/cron/queuedefs` :"

```
r.1j
```

Of course you do not have to take the letter 'r'. You can also choose any other queue name that is not present in the file `/var/spool/cron/queuedefs`.

2.2 User Installation

1. For setting up the user environment the following lines should be added to your own `.cshrc` file:

   ```
   setenv RBS_DIR /home/rous/RBS
   set path = ( $RBS_DIR $path )
   source $RBS_DIR/setup
   source <yourfile>
   ```

   In the last command `<yourfile>` is the path of the RBS environment file (path including the file name) which has to contain the following commands (Notice that the command arguments are examples and should be replaced by your own environment.) :

   ```
   rbs_env              <yourfile>
   rbs_archive          /home/apeall/archive
   rbs_release          /Apeall
   rbs_user_root        /home/rous
   rbs_user             /user
   rbs_batch_machine    prles5
   rbs_batch_queue      i
   rbs_editor           "textedit -font 6x13"
   rbs_deditor          "textedit -font 6x13"
   rbs_root_makefile    makefile.make
   rbs_root_makecomp    general
   ```

In the table below these commands are explained.

`rbs_env`	the path to the current fil
`rbs_archive`	the path to your archive
`rbs_release`	the relative path to the re
`rbs_user_root`	the path to your root dire
`rbs_user`	the relative path to you user
`rbs_batch_machine`	machine that executes integrate
`rbs_batch_queue`	name of the integrate queue
`rbs_editor`	prefered software sources e
`rbs_deditor`	prefered document editor
`rbs_root_makefile`	root make file name
`rbs_root_makecomp`	name of the component in the make file resides

If you work with several archives and several releases you can have more than one file with environment settings. Switching from one environment to the other can be done with the command:

```
source <yourfile>
```

2. Make sure you select the group id that was used to install the archive:

   ```
   newgrp <group id>
   ```

3. Setting up the directory structure

   - Start with sourcing the .cshrc file:

     ```
     source .cshrc
     ```

     (or logout and login again)
   - Create the directory that you specified in the environment file as the `rbs_user`.
   - If you are convinced you specified the RBS environment file correctly, you can issue the command

     ```
     install_user
     ```

     After this command, a complete directory structure exists and the RBS commands can be used.

4. Make sure that you can perform a rsh command (without requiring a password!!) to the machine that you specified in the environment file. Add the machine from which you issue the `integrate` command together with your user id to the `.rhosts` file.

3 Logicals

RBS uses so-called *logicals* to define search paths from the user directories to the archive directory. The file in which these logicals are defined is `RBS_logicals` in the release directory of the archive. This file is created at installation time. It contains an entry for each component in the release. An entry for component X in this file looks like:

```
X    <path1> + <path2> + <path3>
```

This entry specifies that if a file `X` is to be looked up by RBS, the directories `path1`, `path2` and `path3` have to be inspected in this order. The default is that

`path1` corresponds to the source directory of the user.
`path2` is the target directory of the user.
`path3` the the component directory in the archive.

It is possible for the archive manager to specify additional paths for selected components. These paths may point to components in release directories of other archives. For archive consistency reasons this is only possible for a release and all the users of this release: not for individual users. In RBS terminology these archives are called *foreign archives* and RBS provides some commands to use the information in these archives.

4 Make files

4.1 File specifications

In the RBS environment file one has to specify what the location and the name of the current make file is (cf. the section on User Installation). File specifications in an RBS make file always have a fixed form. A file is to be specified as: `<component name>/<file name>`. This means that in the make file one has to abstract from the exact location of the file. RBS uses the search path defined in `RBS_logicals` to locate the file before the actual make starts. In other words, before a build action the make file is expanded in accordance with the actual situation, i.e. the actual location of all files. There is one exception to the form of the file specification: the `integrate` target. A make file, that is, a root make file together with all its includes, must contain exactly one target with the specification:

```
integrate
```

The integrate dependency defines which software is to be build and transferred to the archive during the integrate action.

As explained in the introduction the system distinguishes domain dependent and domain independent components. RBS provides a built-in macro which can be used for files which appear in all DD components. In stead of

```
<component name>/<file name>
```

one can also specify

```
$(domain)/<file name>
```

.

4.2 Dependencies

The dependency between source and target files has to be specified in a format similar to the one used for the standard UNIX make utility:

```
<target_comp>/<target> : <source_comp_1>/<source_1>  \
                          ..........
                         <source_comp_n>/<source_n> ;
<TAB>"action line"
```

In the action line one specifies how the target is to be generated if it is out of date. We advise to use the following format for the action line:

```
@actions/<script> <component> <file>
```

This line means that there is a file `<script>` in component `actions` which is to be executed with two parameters: the component and the file which has to be generated. In general the file is specified without extension since the specified script already uniquely determines the extension. The following dependency is an example of an object file depending on a C source and some include files:

```
tools/test.o : tools/test.c \
               system/globaldef.h \
               unix/system.h ;
        @actions/c tools test
```

The script `c` in component `actions` specifies in detail how the file `test` is to be compiled. In section "Action files" we will give more information about these scripts.
It will be clear that if one of the sources of a certain target is a domain dependent file the the target is also domain dependent, so the above dependency could look like:

```
$(domain)/test.o : tools/test.c \
                   system/globaldef.h \
                   $(domain)/rules.h \
                   unix/system.h ;
        @actions/c tools test $(domain)
```

Notice that the script `c` in the actions line gets as extra parameter value the name of the specific domain, implying that the target file has to be placed in the target component of the actual domain dependent component.

4.3 Include files

A make file may contain include specifications. Such a specification should have the following format:

```
!include <component_name>/<file_name>
```

RBS will try to find the actual location of the include file whenever necessary. The maximal nesting level of include files is 5.

4.4 Miscellaneous

It is also possible to add lines to a make file which obey the unix make syntax, e.g. macro definitions. These lines should be preceded in the first column with a star '*'.

5 Action files

An action file is a UNIX script which specifies how a target is to be generated. The writer of the script can assume that the script is to be executed in the release directory of the user. The script writer is responsible for the fact that all error messages generated by the compiler are written to the file `warnings` in the release directory. RBS provides some tools that can be used by the script writer to find actual locations of files etc.
The following script is an example of a script (with comment) that is to be used to generate an object file from a C source file:

```
--------------------------Example--------------------------------
#! /bin/sh -e
#
# set a trap which will be executed if the script is interrupted
# at an unexpected moment.
#
trap 'echo "c: Error during compilation" ; rm -f $2.c; exit 1' 0
#
# add a newline to the file warnings
#
echo "  " >> warnings
#
# write to standard output that the compilation has started
```

```
#
echo "cc: $1:$2"
#
# try to locate the source file. The RBS utility "get_fname"
# returns the full path name of the file it it has been located,
# otherwise it returns the empty string.
#
src = `$RBS_DIR/get_fname RBS_logicals $1 $2.c`
if test -z "$src"
then echo "cc: Could not find file $2.c"
     exit
fi
#
# copy the source file to the release directory to prevent the C
# compiler from looking for include files in the directory where
# the source file has been found.
#
rm -f $2.o
cp $src .
#
# execute the compilation command. Specify with -I the include
# directiories in the appropriate order, i.e. first source
# directory, next target directory and finally archive directory.
# Since include files may also be found in other components,
# these directories have to be specified as well. Again, take
# care that the compiler error messages are appended to the
# care that the compiler error messages are appended to the
# "warnings" file
#
cc -g -c -D_NO_PROTO -DSUN4 \
    -I$1/source -I$1/target -I$RBS_ARCHIVE$RBS_RELEASE/$1 \
    -Isystem/source -Isystem/target \
    -I$RBS_ARCHIVE$RBS_RELEASE/system \
    -I/cadappl/X11R4/usr/include \
    -I/usr/local/parr/openwin/include \
     $2.c -lm 1>> warnings 2>&1
#
# remove the temporary copy of the include file and move the
# object file to the target directory of the specified component.
#
rm -f $2.c
if test -s $2.o
then
  rm -f $1/target/$2.o
  mv $2.o $1/target/$2.o
fi
#
# reset the trap
#
trap 0
#
#
----------------------------------------------------------------
```

RBS provides a utility called `extract` which looks in the current make file for a dependency of the specified target file and which extracts from this dependency all correctly expanded source file paths. The exact specification is:

```
$RBS_DIR/extract <domain> <file>
```

The output of this command is written to standard output. The `<domain>` parameter specifies what the domain is for which the make file has to be expanded. In the current version of RBS it is assumed that a file name is unique for all targets in the make file, i.e. if a file name occurs somewhere as a target it will occur in no other targets. The following file shows how this utility can be used for linking a number of object files into an executable.

```
------------------------------Example---------------------------

#! /bin/sh -e
#
# set a trap which will be executed if the script is interrupted
# at an unexpected moment.
#
trap 'echo "opt: Error during linking" ; exit 1' 0
#
# write to standard output that the link action has started
#
echo "opt: $1:$2"
#
# create the file `linkfile' which eventually will contain a
# script which executes the actual link.
#
rm -f linkfile
echo "#! /bin/sh -e" > linkfile
echo "cc -Bstatic -o $1/target/$2.exe \\" >> linkfile
#
# extract from the make file all object files which are needed to
# link the executable. Append the actual file names of the files
# to ‘linkfile’.
#
$RBS_DIR/extract $1 $2 >> linkfile
#
# redirect standard error to the file ‘warnings’
#
echo "1>> warnings 2>&1" >> linkfile
#
# make the linkfile executable and execute it
#
chmod u+x linkfile
echo "  "  >> warnings
linkfile
#
# reset the trap
#
trap 0

----------------------------------------------------------------
```

6 General Usage

Suppose we already have a working system in the archive and we want to add a new file to it. The first action we have to undertake is the creation of the new file in the archive, this can be done with the `create` command:

```
create <component> <newfile>
```

Next, we are going to grab this file from the archive. Of course, we want to lock the file to prevent other users from working on the same file. Both operations are performed by the `grab` command:

```
grab <component> <newfile>
```

Now, the file is in our local work space and we can edit it:

```
modify <component> <newfile>
```

If we are satisfied with the contents of the file, we have to connect the file to the system. We do so by changing and adding the appropriate dependencies in the make file. So, first we grab and modify the make file:

```
grab <component> <makefile>
modify <component> <makefile>
```

Suppose the system consists of one executable `system.exe` in component `general`, then we can now try to build this executable in our local work space:

```
build general system.exe
```

With the command

```
sholog
```

we can see what the build operation is actually doing. The trace of previously executed commands can be seen with

```
sholog <nr>
```

where `<nr>` specifies the trace `<nr>`-1 builds ago.
If the build did not succeed, e.g. because of compilation errors, we can inspect the file with warning and error messages with the command:

```
showa
```

We can repeat this sequence of modification and building until we are convinced that everything is correct. The final step is to put the local software into the archive and to remove the lock on the grabbed files. For that purpose we can use the `integrate` command:

```
integrate
```

The integrate command executes a remote shell in order to make sure that the software you developed does not depend on your local environment. Furthermore, it removes all generated targets from your target directories. Next, it tries to build the system again according to the specifications in the make file. If this succeeds, it will move your local software, sources and targets to the archive.

7 Mutual Exclusive Operations

In order to guarantee the consistency of the software in the archive the integration actions have to be scheduled. We can distinguish two subactions within the integration actions:

1. building the 'integrate' target
2. moving the software to the archive.

It will be clear that the archive will become inconsistent if we would allow one RBS user to build the `integrate` target while another user is moving software to the archive. During the transfer of software the archive is temporarily inconsistent. Therefore, RBS locks the archive for builds and integrates of other users during this period. This is not the only problem, however. If two users would do phase 1. in parallel and phase 2. sequentially, the archive would also become inconsistent because the build of user A does not take into acount the changes of user B and vice versa. In order to overcome this problem integrate actions have to be performed sequentially. This is the reason that a special integration queue has to be defined which allows only one job at the time to be executed. A third way to make the archive inconsistent would be if user would be able to perform `grab`, `free`, `modify` or `build` actions during an `integrate`. As a consequence, RBS does not allow to execute these operations in parallel. With respect to the local workspace it is the RBS philosophy that this is the responsibility of the user. He can do whatever he wants as long as he uses the RBS procedures to put software into the archive[1]. There is one exception: it is not allowed to start a new build while your previous one has not yet finished.

RBS handles these mutual exclusive operations with lock files. A weak point in this implementation is the fact that if an operation is killed in an unexpected way the lock file is not removed. As a result RBS does not allow other operations to be executed. A user can repair this by removing the lock files in question with the `rmlock` command. The user can use one of the following parameter values for this command: `modify`, `build`, `free`, `grab` or `integrate`, depending on the action that crashed.
Since RBS cannot check whether the `rmlock` utility is used correctly by the user, an appeal is made to his sense of responsibility.

8 Security

RBS uses the UNIX facility to define 'groups'. All users in the same group have the right to change the archive. Of course it is assumed that the archive will only be changed with the 'integrate' action. RBS cannot prevent users in a group to actually alter or delete files directly in the archive. Assuming they they don't do these things on purpose, the only thing RBS can do is to provide enough functionality for a user, so he doesn't feel the need to perform UNIX shell operations directly on archive files.
Users which are not in the group have read privilige on all files in the archive.

9 The Ten RBS Commandments

1. Thou shalt use only RBS commands to operate on RBS files and directories.
2. Thou shalt not use RBS commands in an improper way.
3. Thou shalt not try to be smarter than RBS.
4. Thou shalt not forget to keep your make files consistent with your software.
5. Thou shalt only covet thy neighbour's files via the archive.

[1]Of course we highly recommend to use only RBS commands.

6. Thou shalt not repair the result of offences against the commandments with new offences. Thou should warn the RBS manager immediately.
7. Thou shalt accept that RBS cannot fulfill all your desires.
8. Thou shalt not try to change the RBS scripts
9. Thou shalt treat the RBS developer kindly since he is always right.
10. Thou shalt always obey the ten RBS commandments and especially the tenth one.

10 Making a document

This document is called rbsmanual.tex. To create it I gave the command

```
dcreate rbsmanual.tex
```

The command dcreate asks for the title and provides the number of the document. This number could be used for you project document administration. the `dcreate` adds the name of your document, its number, and its owner to a file, which you can inspect by typing

```
dsurvey
```

If you do it you will see to see that this document indeed exists. You can type

```
dinspect userbs.tex
```

to see (the latex source of) my document, as it is in the archive.
If you have 'dcreated' a file, say myfile, or if you want to change a document that is in the archive, say

```
dgrab myfile
```

Now a copy of the file comes in your document source directory. Only one person can grab a file. If someone else has it, you are notified. If 'myfile' is a text file, you can modify it with

```
dmodify myfile
```

(or `modify doc myfile`) which opens the file with your favorite texteditor (specified in your environment file). This command works in any directory, so you will not have to know its exact location. If it is a framemaker document, there is no such command, so you must know the directory. If you want to use framemaker for a new file, you must overwrite the empty file that is created by the first dgrab. This means that you must perform the command

```
cdsource doc
```

This is not so for latex commands.

The command `dbuild` can be used from any directory:

```
dbuild file.dvi
```

builds the `dvi` file from `file.tex`. The first time it will run latex three times. After that, it will run latex *exactly* as many times as needed to get cross-references right. Likewise,

```
dbuild file.preview
```

makes a preview of the file. If the `dvi` file is not up to date it will first execute latex to make it up to date. This also works if the `tex` file is in the archive. One could change the file doc.make such that the command

```
dbuild file.print1
```

prints the file two-sidedly (after making the dvi file up to date), on the first floor,

```
dbuild file.print2
```

does the same on the second floor. If 'file.fm' is a framemaker document, then

```
dbuild file.fmview
```

starts framemaker with the file. If the file has been grabbed this is the same as `dmodify file.fm`, if your environment file states that your document texteditor is framemaker. If the file is in the archive `dbuild file.fmview` also works, but modifications made to the file are ignored.
For other latex commands (bibtex), or another previewer, or for printing your file somewhere else, other targets can be added in a very simple way (by *you*: in the file doc.make in component *doc*).
When your document is finished you may say

```
dintegrate
```

This command is a bit rough. It moves all grabbed document files to the archive. It mails to all group members (the ones that have installed themselves as RBS users) which documents have been integrated. Thus, if you get a mail that a new version of file `myfile.tex` has been integrated, you can type

```
dbuild myfile.print1
```

to send it to the printer at the first floor. If it is 'file.fm', a framemaker document,

```
dbuild file.fmview
```

loads the file in framemaker so that you can print it.

11 RBS commands

11.1 Most often used commands

`build <component> <file>`	build target `<file>` belonging to component `<component>`. All results (final and intermediate are placed in the "target" directory of the corresponing component. The build progress can be inspected with the "sholog" command (see below). The error messages produced by compilers and linkers can be inspected by the "showa" command.
`create <component> <file>`	creates a new file `file` in the specified component in the archive.
`free <component> <file>`	releases the lock on the file in the archive and removes the file from the source directory!
`grab <component> <file>`	tries to lock `<file>` in `<component>` in the archive and puts a copy into the user's source directory if the lock succeeds.
`inspect <component> <file>`	put a copy of the corresponding file in the archive in an edit window.
`integrate`	cleanup the user components (cf `cleanup` command), build the integrate target (specified in the makefile) and move all files to the archive, if the build succeeds.
`modify <component> <file>`	edit the corresponding file in your source directory.
`newa`	throws away all old compiler error messages.
`sholog <nr>`	if `<nr>` is empty or equals 1 the command shows the results of the last (current) build, higher numbers show previous build results. RBS will only remember the results of the 10 most recent "builds".
`showa`	shows the error messages produced by `build` commands.

11.2 Some more information-providing commands

`allsources`	shows all source files that have been grabbed by you.
`alltargets`	shows all files that have been generated by you in the target directory.
`allsaves`	shows all files that have been placed by you into the save directory.
`shoenv`	shows the settings of the current RBS environment.
`archs <component> ["pattern"]`	gives an overview of all files belonging to the specified component in the archive. Only the files that match the specified pattern will be presented.
`sources <component> ["pattern"]`	gives an overview of all sources belonging to the specified component that have been grabbed by you. Only the files that match the specified pattern will be presented.
`targets <component> ["pattern"]`	gives an overview of all targets belonging to the specified component that have been generated in your own local directory. Only the files that match the specified pattern will be presented.
`saves <component> ["pattern"]`	gives an overview of all saved files belonging to the specified component that have been generated in your own local directory. Only the files that match the specified pattern will be presented.

11.3 More file-manipulaton commands

`cleanup`	removes all files from all target directories, checks if all source files are legally grabbed and removes all junk files from the source directory.
`rmlock <command>`	it can happen that lock files remain after a command has been killed externally. In that case the lock files created by the specified command can be removed manually with "rmlock".
`tcopy <component> <file>`	makes a copy of the corresponding file in the archive and puts it into your target directory. Notice the difference with the grab command.
`tmodify <component> <file>`	edit the corresponding file in your target directory.
`tosave <component> <file>`	moves the specified file to a "safe" place, in which it cannot be found by a build or integrate.
`tosource <component> <file>`	moves the specified file from its "safe" place to the source directory.

11.4 Commands related to other users

`info <component>`	shows which files are locked in the specified component.
`hisint`	gives a history of the integrate actions.
`shoiq`	gives an overview of the integrate jobs in the integrate queue.
`whbusy`	shows which users are building or integrating.
`whgrab <component>`	shows which files of the specified component are grabbed.

11.5 Commands related to other releases

`sdrag <fromrelease> <component>`	move all files from the source directory of the `<component>` of the specified release (the full path to the release has to be specified) to the source directory of the `<component>` of your current release.
`setup <archive> <release> <user>`	specify the archive, the release and the user directories by means of (relative) pathnames.
`tdrag <fromrelease> <component>`	move all files from the target directory of the `<component>` of the specified release (the full path to the release has to be specified) to the target directory of the `<component>` of your current release.

11.6 Commands to quickly go to RBS directories

`cdarch <component>`	go to the directory `<component>` of the archive.
`cdrelease`	go to the release directory of the user.
`cdactions`	go to the `actions` directory of the user.
`cdsource <component>`	go to the source directory of `<component>`.
`cdtarget <component>`	go to the target directory of `<component>`.
`cdsave <component>`	go to the save directory of `<component>`.

11.7 Commands related to foreign archives

`fgrab <component> <file>`	creates a new file in the archive, grabs the newly created file and instantiates it with the contents of the corresponding file in the foreign archive.
`finspect <component> <file>`	view the file in the foreign archive.
`fsetup <archive> <release>`	specify the archive and release of the foreign archive by means of (relative) pathnames.
`ftcopy <component> <file>`	makes a copy of the corresponding file in the foreign archive and puts it into your target directory. Notice the difference with the "fgrab" command.

11.8 Document Commands

`dcleanup`	removes all files from the doc target directory, checks if all source files are legally grabbed and removes all junk files from the source directory.
`dbuild <file>`	Performs actions according to the file doc.make in component doc.
`dcreate <file>`	creates a new document file in the archive and gives the document number belonging to the file. The user is asked to specify the title of the document.
`dfree <file>`	removes your own document file and releases the lock in the archive.
`dgrab <file>`	puts a copy of the archive file in your doc directory and locks the corresponding file in the archive.
`dinspect <file>`	inspect the last version of the corresponding file in the archive.
`dintegrate`	copy all grabbed (!!) user documents to the archive.
`dmodify <file>`	modify the specified document.
`dsurvey`	give a survey of all numbered documents.