©1996–2001 Harald Kirsch

bras
another kind of ‘make’

User Manual & Reference

Harald Kirsch
pifpafpuf@gmx.de

March 4, 2002

Contents

1 User Manual

1.1 Introduction

If you ever used `make`, you may have noticed that it has a few odd features. Most annoying to me are

- the crazy syntax (tabs as a structuring element),
- the way variables are expanded (I never found out, when exactly this happens),
- the inability to call `makefiles` in other directories without breaking the chain of reasoning [Miller 1997],
- the inability to distinguish between targets that merely need to exist, those that must be newer than their dependencies, and others which must be made for totally different reasons,
- the inability to call the reasoning process explicitly without forking a new `make`, thereby breaking the chain of reasoning,
- the lack of control structures,
- the lack of decent pattern matching.

It is indisputable that **the** feature of `make` — rule based command execution — is most helpful in many situations; in particular were it seems to be applied most often: conditional compilation of large programs.

My idea was to combine the key features of `make` with all the niceties of a command language, while improving on the misfeatures mentioned above. `Bras` uses user-supplied conditions to test if a target must be made. Most of the time they just test, if the target is older than some other files, like `make` usually does, but totally different tests are possible.

Since I personally like Tcl, not the least for its well defined handling of command line substitution (see `Tcl(n)` and no flames, please), I ended up with the present solution. A similar solution might be possible with other command languages.

1.2 Rules

1.2.1 Syntax

Unlike in shell-scripts or programming languages like C, Ada or Java, the order of command execution in `brasfiles` is not only described with control-structures like if-then-else, loops, etc., but also by rules. The main ingredients of a rule are explained along the following example.

```
Make fft.o { [older fft.o {fft.c fft.h}] } {
  cc -o fft.o fft.c
}
```

The Tcl-procedure `Make` is used to register a rule in the rule-database. In fact, `Make` ist very similar to the Tcl-command `proc` in that it only declares or registers some code in the Tcl-engine rather than evaluating anything. Only later, if `bras` is asked to reason about the file `fft.o`, the rule becomes active and actually does something.

`Make` has three parameters which make up the rule:

target The file `fft.o` is called the *target* of the rule. The ultimate goal of writing a rule is to describe under which conditions the target must be made and how this is done.

condition The second part of a rule is called the *condition*. In the example, the condition is

```
[older fft.o fft.c fft.h]
```

The condition must be a Tcl-expression very similar to those used in Tcl's `if`-command. Whenever `bras` is instructed to build the target `fft.o` it first evaluates the condition to find out if it is really necessary to make the target. If evaluation of the conditon returns *true*, the target is considered *out-of-date* and must be built.

command The third part of a rule is called the *command*. In the example it is the script

```
cc -o fft.o fft.c
```

It can be more or less any piece of ordinary Tcl code. However, commands not known to Tcl are assumed to be commands of the operating system and are passed to `[exec]` for execution (see 1.12.4 for details).

The command is executed only if the condition is true. `Bras` assumes that the target is up-to-date after the command has been executed. It does not check if this is actually true.

To tell `Bras` explicitly that no *command* needs to be executed, use the string `.relax.` possibly surrounded by white space (see 2.3.1).

The general form of a rule is:

`Make` *targets condition command*

and its meaning can be described as

Make the *targets* with *command* if *condition* is true.

The first argument of `Make` can be a list of targets. `Bras` assumes that all of them are made whenever the rule's command is executed.

Most of the power and flexibility of `bras` depends on the fact that the condition rendering a target out-of-date can be just any boolean expression. Ancient `make` on the other hand knows just one condition: the target does not exist as a file or it is older than some other files.

1.2.2 Conditions

As said before, a rule's condition can be just any boolean expression. These expressions however are usually a bit more elaborate than what is most often used in `if`-statements. For example to simulate `make`'s behaviour, the boolean expression must check if a file exists and if it is older than a bunch of other files. This is not a test readily available in Tcl, but it can be implemented as a Tcl-procedure.

The procedure `older` used in the example of the last section implements this test. In the rest of this document procedures which test something and return a boolean result are called *predicates*. There are several other predefined predicates as well which test certain conditions or relations between files. They are described in appendix 2.1.

A few more words must be said about the execution environment or variable context whithin which the condition of a rule is evaluated. Consider the following rule:

```
Make {a b c} {[older $targets X] || [missing $target]} {
  do something
}
```

Its condition tests if any of the targets `a`, `b` or `c` is older than some other file `X` or if the target for which the rule was invoked does not exist as a file. The rule demonstrates the use of the variables `targets` and `target` which are set just before the condition is evaluated:

`targets` will be set to the list of the targets of the rule which happens to be `{a b c}` in this example,

target will be set to the target for which the rule was actually invoked. It can be any of the targets of the rule.

Most of the time, the condition contains just one predicate or is a logical combination of a few predicates. The followingollowingollowingollowingcates incates incates in

1.2.3 Predicates

Predicates are test-procedures implementing the conditions under which a target is up-to-date. The standard predicates of `bras` are described in appendix 2.1. Here we describe how a predicate works and how new predicates can be written.

Consider the example

```
set OBJ [list a.o b.o c.o d.o e.o]
Make libX.a {[older $target $OBJ]} {
  ar r $target $trigger
}
```

The rule is used to replace those files of an archive `libX.a` which are newer than the archive, i.e. which trigger the command to be executed.

But how is variable `trigger` initialized? That is a service of the predicate `older`.

In general a predicate can register a list of variables with the reasoning engine to be passed to the command. It depends on the predicate, which information might be useful for the command. However these variables are used by most predicates:

deps is the list of files (or other objects) on which the target — or the predicate test — depends, i.e. these files must themselves be up-to-date before the predicate can perform its test. In fact it is the responsibility of the predicate to make sure, they are up-to-date. Read on to find out how this is easily done.

trigger is a list of files (or other objects) which are particularly responsible for the target being out-of-date. For the `older`-predicate `$trigger` contains all files which are older than the target.

Now lets look at an example predicate, namely `true`, which always returns a non-zero value. Its code is short but demonstrates all that is necessary for a good predicate. To see more examples have a look at the file `predicates.tcl` which is part of `bras`.

```
1     proc ::bras::p::true {{inDeps {}}} {
2       installPredicate [list deps] inDeps
3
4       ::bras::consider $inDeps
```

```
5
6       append reason "\nmust always be made"
7       ::bras::concatUnique deps [stripAt $inDeps]
8       $inDeps return 1
9     }
```

First of all, every predicate should be defined in the namespace `::bras::p`. If you choose a different namespace, you are on your own. It might work or not.

A good predicate should as its first task call `installPredicate`, which takes two parameters:

1. A list of variables to be declared such that they end up visible in the execution environment used to execute the rule's command. The `true`-predicate wants to pass variable `deps` to the rule's command. Passing variables is implemented by linking the variable names via `upvar #0` to the namespace used for the command (which is **not** `::bras::p`).

2. A list of variable names each of which contains a list of dependencies. These dependencies are expanded by `installPredicate` along the `searchpath` (cf. section 1.8) and the expansions are put back into the variables. The `true`-predicate has just one such variable, namely its parameter `inDeps`.

Next, the predicate should usually call `::bras::consider` and pass it the list of all those targets which must be up-to-date before the predicate can perform its test. Although the `true`-predicate does not actually perform any test, it takes a list of dependencies as its first argument for the sole purpose of running them through the reasoning engine. This is useful to combine a list of targets into one (pseudo)-target like in

```
Make all {[true all-libs all-docs]} {
   puts "successfully made $deps"
}
```

where target `all` triggers the built-process for `all-libs` and `all-docs`.[1]

The procedure `::bras::consider` returns a list of zeros and ones with one entry for every dependency. The result indicates that a dependency was made anew if and only if the respective return value is 1. Predicate `older` uses this in its test, but `true` does not need the result.

Before the predicate returns the result of its test, it has two more responsibilities.

1. It has to append an informative string to the variable `reason` with a leading `\n`. This string is printed when `bras` is called with option `-d`.

[1] A better way to combine several targets under one name makes use of the predicate `updated`, described in section 2.1.9.

2. It must not forget to update the variables registered with `installPredicate`. Because more than one predicate may be called during evaluation of a rule's condition, the predicate must be careful not to overwrite these variables and should usually append to them. Currently there is no safe way to register a private variable for a predicate to be passed to the rule's command.

The return value of a predicate must always be a boolean value.

1.3 Multiple Rules for a Target

It is not uncommon to have more than one rule for a target. In particular when dependency relations are computed automatically, it is not always possible to pool them into one rule. However one condition must be met by the set of rules mentioning a certain target: at most one of them may have a non-empty command.

An example for the use of more than one rule is

```
Make a.o {[older a.o {b.h a.h}]}
Make a.o {[older a.o a.c]}
Make a.o {[older a.o a.h]}
```

Internally, `bras` combines all rules for a target into a single one. The conditions of all the rules without a command are combined by a logical non-short-circuit disjunction (*or*), i.e. they will all eventually be evaluated one after another and then the results are combined with `||`.

As mentioned above, only one of the rules may have a command. This rule plays a special role in that `bras` arranges for its condition to be evaluated first. In effect the predicates of that condition are the first to enter values into variables like `deps` which are passed to the combined rule's command. Consequently, idioms like `[lindex $deps 0]` can be used in the command to access just the right dependencies.

Example:

```
Make a.o {[older a.o {b.h a.h}]}
Make a.o {[older a.o a.c]} {
  cc -o $target -c [lindex $deps 0]
}
```

The condition of the second rule will be evaluated before the condition of the first one. As a result `[lindex $deps 0]`will be `a.c` and not `b.h` as it would be if the evaluation order was not reverted.

1.4 Multi-Target Rules

Sometimes, a rule's command generates more than one file (target). A typical example is the compilation of several Java source files with one call to the compiler (for `pairedolder` see section 2.1.7):

```
Make {A.class B.class} {
  [pairedolder $targets {A.java B.java}]
} {
  javac $deps
}
```

Whenever `bras` finds several targets in the same rule,

> it assumes that the predicate tests all targets at the same time,

i.e. it suffices to evaluate the predicate once. Internally, `bras` never considers more than one target at a time. However, if one target of a multi-target rule is considered, the result of evaluating the predicate is distributed over all targets of that rule. In particular:

- If the predicate says that one target is up-to-date, all of them are marked up-to-date.
- If the predicate says that target is out-of-date, the command is run and afterwards it is assumed that all of the rule's targets are up-to-date.

Extending the example above with the rule

```
Make classes {[updated {A.class B.class}]} .relax.
```

consider what happens when

```
bras classes
```

is called. The predicate `updated` checks `A.class` and `B.class` in turn. Consideration of `A.class` will immediately mark `B.class` as either up-to-date or "just made". When checking `B.class`, `bras` does not run `pairedolder` again — nor the compilation — but (re)uses the result already derived for `A.class`.

Note: handling of multi-target rules is completely different from GNU `make`. According to the documentation, (at least GNU) `make` interpretes a multi-target rule as if it is a shortcut for writing the same rule for each of the targets.

1.5 Pattern Rules

Pattern rules extend on the idea of `make`'s suffix or implicit rules and serve two purposes:

1. If there is no rule for a target under consideration, `bras` tries to find a pattern rule and uses it to create a rule on the fly. The matching process is described in section 1.5.1.
2. If a rule with an empty command is triggered, `bras` tries to find a pattern rule to substitute its command as a default.

The general form of pattern rules is

`PatternMake` *regexp deptag condition command*

The individual elements of a pattern rule are:

regexp is a regular expression used to select targets to which the rule applies,

deptag is used to aid the process of selecting the correct pattern rule. The meaning of the *deptag* is described in the following sections.

condition is a test in the same way as for explicit rules,

command is the command to be used in a derived real rule or as a default command.

The details about *regexp* and *deptag* are described in sections 1.5.1 and 1.5.2.

An example pattern rule to translate TEX's DVI-files into PostScript looks like

```
PatternMake .*\.ps .dvi {[older $target $d]} {
  dvips -o $target [lindex $deps 0]
}
```

Note the use of variable `d` in the condition. It is a special variable only available in the condition of a pattern rule. In the example it will contain the file name `bla.dvi` whenever the pattern rule is used to build the target `bla.ps`. See section 1.5.3 for more details.

When the condition is evaluated, `$target` will be set to the name of the target under consideration and `$deps` will be set to the list of its dependencies, with `$d` as its first element. The idiom `[lindex $deps 0]` is similar to `make`'s automatic variable `$<` and expands `$d`.

1.5.1 How pattern rules are used to create real rules

As mentioned above, pattern rules are used to create a new rule if there is no explicit rule available for the target under consideration. For a given target, a pattern rule is selected as described below.

All known pattern rules are checked in the opposite order in which they were specified so that rules defined later override those defined earlier.

For a pattern rule to be a candidate at all, the given real target must match the pattern rule's regular expression. To make sure that the whole word matches, regular expressions of pattern rules get an implicit `^` and `$` prefixed and suffixed. Put another way: if `target` is the variable containing the target under consideration and `rexp` is the regular expression of a pattern rule, `bras` executes an equivalent of

```
regexp "^$rexp\$" $target
```

to check whether the pattern rule is a *candidate* or not.

Suppose a candidate pattern rule is found for a target, e.g. `bla.o`. It must then survive the following two-step test to be taken.

Step 1 derives a dependency from `bla.o`. It calls a procedure with a name derived from *deptag*, namely

```
::bras::gendep::deptag  $target
```

to create a dependency name. For example if *deptag* is `.c`, `bras` calls

```
::bras::gendep::.c $target
```

to derive a dependency name. The default `::bras::gendep::`*`deptag`* looks like

```
proc ::bras::gendep::deptag {target} {
    return [::bras::defaultGendep $target deptag]
}
```

The procedure `::bras::defaultGendep` strips the file extension from the target and suffixes the result with *deptag*. To change the behaviour of individual dependency generators, it suffices to define it in the namespace `::bras::gendep`. To change the default behaviour of all dependency generators, redefine the procedure `::bras::defaultGendep`. Note however, that this might break some default rules delivered with `bras`.

Step 2 subjects the dependency name to `searchpath` (see section 1.8). If it finds the dependency name as an existing file, or if it finds a real rule for the dependency, the pattern rule is taken.

If a pattern rule is taken, `bras` creates a real rule with

- the target under consideration as the rule's target,
- the given condition as the condition of the new rule and
- the command of the pattern rule as the rule's command.

It then proceeds as if there had never been a search for a pattern rule.

If no rule is selected after checking all pattern rules as described above, `bras` tries one other thing. For each rule which matches the target, a dependency is derived as described in step 1. Then this is taken as a new target and `bras` calls the above pattern-rule selection recursively. It is made sure, that the recursion does not loop infinitely by not trying any pattern-rule on different recursion levels at the same time.

Ultimately, `bras` will either run out of pattern-rules, in which case it decides that there is no rule for the target, or it derives a dependency which either exists as a file or has an explicit rule. All intermediate pattern-rules are then converted to real rules and the reasoning proceeds as if there were never any rules missing.

The algorithm is (should be :-) implemented in the file `lastMinuteRule.tcl`.

1.5.2 How default commands are derived from pattern rules

Besides being rule patterns, pattern rules are also used to define default commands. The situation described in the last section is one where there is **no** explicit rule for a certain target. In that case a rule **and** a command are derived.

This section describes the procedure taken if there is a rule for a target, however without a command. A typical situation is described by the following rules.

```
Make a.o {[older a.o b.h]
Make a.o {[older a.o a.c]
Make a.o {[older a.o a.h]
```

A single target, `a.o`, depends on several dependencies which are spread over several rules. None of the rules defines a command to build the target. To find a command, `bras` also uses the pattern rules. Without taking into account the dependencies already known, `bras` uses the same algorithm as described in section 1.5.1 to find a pattern rule for the target. If a rule is found, its condition **and** its command are added the target's rule as described in section 1.3[2].

In particular the new condition is put in front of all conditions already known for the rule. Because it will be evaluated first, its dependencies will be in front of `$deps` when the command is executed, which is usually a good thing. Read section 1.5.3 for further details.

[2] algorithm stolen from `make`

1.5.3 Accessing the generated dependency in the condition

As described in section 1.5.1, a pattern rule is only choosen for a given target if also the dependency *d* derived with the dependency tag is either an existing file or has an explicit rule. This dependency must usually be used in the condition, but when the condition is written down, its name is of course not yet known. Consequently, the name must be passed in a variable. `Bras` arranges for that dependency name to be available in the variable `d`, when the condition is executed. An example use of that variable can be seen in the following pattern rule:

```
PatternMake .*\.o .c {[older $target $d]} {
  $CC -o $target [lindex $deps 0]
}
```

If the rule is choosen for a target `bla.o` and its derived dependency `bla.c`, `$d` will expand to `bla.c` when the condition is executed.

1.6 Dependencies in Other Directories

Most annoying to me when using `make` is its unwieldy handling of dependencies in another directory. Calling `make` recursively after changing to the foreign directory often breaks the chain of reasoning, because there is no communication between parent- and child-make as to whether any target was build or whether all targets were up-to-date. Have a look at [Miller 1997] to understand why `make`'s behaviour in this area is normally not what you want.

`Bras` allows a different solution. Whenever a dependency starts with `@` and does not belong to the current directory, `bras` expects a `brasfile` in the directory where the dependency belongs to. It uses the rules found there to reason about the dependency — if considered as a target. Lets look at an example:

```
Make bla {[older bla.o @../libfasel/libfasel.a]} {
  $CC -c -o $target $CFLAGS $LDFLAGS $deps
}
```

Whenever the command `bras bla` is called, the rule is considered, and consequently all dependencies in turn are considered as a target. In particular `@../libfasel/libfasel.a` tells `bras` that there is a subdirectory `../libfasel` with a `brasfile` which describes how to handle the target `libfasel.a` locally in that directory. `Bras` reads `../libfasel/brasfile`. Then it changes to directory `../libfasel` before executing any commands for rules in `../libfasel/brasfile`.

1.7 Including Files Only Once

As described in section 1.6, `bras` automatically sources the rule file in other directories if an @-dependency leads there. But this mechanism is not very useful to source a file with global definitions. Instead, the Tcl-builtin command `source` could be used, but it has the disadvantage that it does not guard against sourcing the same file twice.

To make sure a file is sourced exactly once, use the `include`-command. It takes as its only parameter the file to read.

Thats the simple truth, but consider the following setup. The main directory of application `bla` has a `brasfile` and also several subdirectories like `lib`, `extrabla`, `nobla`, the targets of which are called from the main `brasfile` by means of @. What happens, if you intend to solely work in directory `lib` for some time? What is the best way within `bla/lib/brasfile` to get access to all those global definitions in `bla/brasfile`?

Using `include ../brasfile` will not work, because all rules of `bla/brasfile` would be interpreted relative to `bla/lib`, which is usually wrong. A workaround might be:

```
cd ..
include brasfile
cd bla
```

But this does not set the namespace correctly for the included `brasfile`. To execute a `brasfile` in the same context as it would be executed when called implicitly by an @-target, use e.g.

```
include @.
```

In general, `include` has one argument which is either a filename or is of the form @*dir*, where *dir* is a directory name. In the latter case, it looks for a `brasfile` (or `Brasfile` or whatever was specified on the command line) and sources the file in the context of the given directory.

Hint: If the included script needs to know its own name, use the standard Tcl-command `[info script]`.

1.8 Searching for Dependencies

In built-environments were source directories are read-only, compiled files must be put into a different directory than source files. If in addition builts for different platforms must be supported, source files may be found in different directories depending on the platform for which the built is performed.

One way to support that would be to create explicit rules in a loop for all source files listed e.g. in variable SRC:

```
foreach x $SRC {
  set base [file root $x]
  lappend OBJ build/$base.o
  Make build/$base.o {[older $target $platform/$base.c]}
}
Make all {[true $OBJ]} {}
```

This has at least two drawbacks:

1. The few lines above will become even more elaborate as soon as a source file may be either platform-specific or generic.
2. Pattern rules are no longer used because all rules are explicitly constructed.

To overcome these drawbacks, the command `searchpath` was introduced. With one argument it registers a list of pathnames which are used to locate dependencies, e.g.

```
searchpath {. ./generic ./unix}
```

Every (builtin) predicate, before reasoning about its parameters, tries to find them along the search path if they don't start with an @ and qualify as

```
[file pathtype ...]==relative
```

In particular it first tries to find a file with that name along the search path. If it does not find the file, it then checks if there is an explicit rule for the name in one of the search paths. The dependency found is then taken as the true dependency. If no file or explicit rule can be found, the first directory in the search path is assumed to be the correct one. Therefore the current directory, i.e. a single dot, should almost always be the first element in the list given to `searchpath`.

The search path declared with `searchpath` is local for a directory, i.e. if `bras` follows an @-target to a different directory, this directory has its own search path.

To query the current search path, use `[searchpath]`.

1.9 Use of Automatically Generated Dependencies

1.9.1 The ancient way

Most C-compilers are able to generate makefile-dependencies for source files by determining which files are included with `#include` directives. For example `gcc` can be

instructed to do so with option `-M` while SUN's Solaris C-compiler uses `-xM`. Because this is a valuable feature, `bras` has the ability to read and understand a restricted type of `make`-dependencies. The command

`sourcedeps` *file*

reads a dependency file created by the C-compiler and creates the appropriate rules. The command prints a warning if the given file does not exist.

1.9.2 The modern way

Q: When must a C source be compiled?

A: If either the file itself or at least one of the files which are directly or indirectly included was modified.

Consequently, the dependency list for an object file must contain the C source file as well as all included files. Editing one of the files of that list can result in the deletion or addition of an included file, thereby changing the dependency structure itself and rendering the dependency list out of date. Therefore a static dependency list which was written down or generated the other day will soon be out of date.

`Bras` allows to update the dependency list as necessary. An example how this can be done for the relation between object files and C source files can be found in the files `c2o.rule` (2.2.1) and `cdeps.rule` (2.2.4). To use them, include them both into your `brasfile` with

```
include $::bras::base/c2o.rule
include $::bras::base/cdeps.rule
```

The mechanism uses the builtin predicates `dcold` and `oldcache` which are described in appendix 2.1. For other programming languages similar schemes can be implemented. (Send me your patches, please!)

1.10 Explicitly Calling the Reasoning Process

Normally the reasoning process is invoked automagically for the first target found in the `brasfile` or for the targets mentioned on the command line. However, sometimes it might be necessary to call it explicitly, in particular within a rule's script. In a `makefile` `make` is `exec`ed recursively in such situations. The disadvantage of this approach is, that results of the reasoning which was already performed by the parent process are not available to the child process. Consequently the child starts all over again. `Bras` allows to call the reasoning process explicitly without `exec`. Just call

`consider` *targets*

to let `bras` consider and update the given target. The result of consider is a list of zeros and ones, one for each target in the argument list. A one is returned if and only if the respective target was made. An error is returned, if one of the targets cannot be made.

1.11 Using Environment Variables

One of the nicer features of `make` is its ability to override variable values in the makefile with values from the command line or from the environment with `var=value` or `-e` on the command line respectively.

A similar feature exists in `bras`. To set a variable in a way that it can be overridden by the environment or on the command line, use `getenv` instead of `set` in your brasfiles. Example:

```
getenv prefix /usr/local
```

This will either set `prefix` to `$env(prefix)` or to `/usr/local` if the latter does not exist. If you require an environment variable to be set, leave out the default as in

```
getenv BLA
```

If there is no variable with name `BLA` in the environment, this will result in a Tcl-error.

1.12 How `Bras` Executes

There are the following major phases of operation performed strictly in sequence.

1.12.1 Set env-entries from the command line

All definitions like `var=value` found on the command line are entered into the global array `env` thereby overriding the respective values from the environment.

1.12.2 Read `brasfile`

The `brasfile` is read and executed. This is very similar to what the `source`-command does. However, sourcing occurs in a dedicated (anonymous) namespace which is tied to the directory in which the `brasfile` is found. Consequently, procedures and variables set in the `brasfile` will end up in that namespace if they are not explicitly put

into another one. This automatically shields the operations of the the `brasfiles` in different directories from each other. For example it allows to reuse the variable name `SOURCES` in every `brasfile` of a multi directory built environment. See `linkvar` (p. 43) and `dirns` (p 41 for ways to communicate information between the directory related namespaces.

The command to define rules, `Make`, **only records** rules in an internal database. In particular, no command specified in a rule is executed.

Beside `Make`, every Tcl-command can be used in a `brasfile`. They are executed as in every other Tcl-script. Useful examples are setting global variables or `if`-statements including or excluding system-specific rules.

1.12.3 Reasoning process

Either the targets given on the command line or the targets of the very first rule found in `brasfile` are considered in turn. Considering a target can have three different results.

1. The target is up-to-date.
2. The target is not up-to-date but there is a way to built it.
3. The target is not up-to-date, but something necessary for building it is missing and cannot be build.

Checking a target is a recursive process. It requires consideration of the target's dependencies. The exact order and interpretation of recursive calls of the reasoning process are described below.[3]

• If the target starts with the character `@`, `bras` changes to the directory of the target and reads the `brasfile` found there. Any directory part is removed from the targets name.

• If the target was considered already along another line of reasoning, it is not checked again. Instead the result computed before is immediately returned.

• If no rule is applicable for the target, `bras` tries to make one up as desribed in section 1.5.1. If none can be constructed, the reasoning process returns immediately with one of two results: If the target denotes an existing file, it reports that the target is up-to-date. Otherwise it reports that the target is not up-to-date and cannot be built.

• If there are rules for the target none of which specifies a command to update the target, a command is constructed as described in section 1.5.2. If no command can be made `bras` merely prints a warning later, if it is decided that the target needs an update.

[3] The most precise description of what `bras` does is in the source file `consider.tcl`. I do my best to match it as closely as possible.

• All conditions of a rule are evaluated in turn and their results are combined with a logical disjunction (or). Predicates called during that evaluation will eventually expand their arguments along the search path and pass them recursively to the reasoning process before they perform their own test. The predicates are executed in the namespace in which the `brasfile` was sourced. Consequently they have access to all variables which are defined in the `brasfile`.

• Finally, the rule's command is executed if the disjunction of the conditions returns a non-zero value. Details on how this is done can be found in section 1.12.4.

1.12.4 Command execution

All commands derived from rules are executed in the namespace of the `brasfile` in which they were defined. This is also true for commands which were derived via pattern rules. Before the command is executed, the namespace will have the following variables set.

`targets` contains the list of targets for that rule,

`target` is the name of the target for which the rule was invoked,

In addition, many predicates set one or both of the following variables:

`deps` is the list of the rule's dependencies (note that `[lindex $deps 0]` is mostly equivalent to `make`'s variable `$<`). It usually depends on the predicate what exactly a dependency is. Usually these are the files or targets which must be up-to-date before a predicate can perform its test.

`trigger` contains those elements of `deps` which rendered the target out-of-date. The contents of this variable again depend on predicate.

After setting these variables and changing to the right directory, a rule's command is executed. Non-internal commands are automatically passed to `exec`. But please remember that `bras` is basically Tcl and unlike `sh`, it does no globbing on the command line. Consequently, the rule

```
Always clean {} {
  rm -f *.o *.a *~
}
```

does not work as expected. You have to resort to

```
Always clean {}
  rm -f [glob -nocomplain -- *.o *.a *~]
}
```

A delicate problem is variable substitution in external commands. Suppose `CFLAGS` is set to `"-g -Wall"` and there is a command like

```
cc $CFLAGS -c [lindex $deps 0]
```

If external commands were only `exec`'ed you could **not** expect this to work, because it were equivalent to the tcl-script

```
exec cc $CFLAGS -c [lindex $deps 0]
```

which calls `cc` with just 3 arguments resulting effectively in

```
cc "-g -Wall" -c whatever.c
```

To make it work, `bras` actually does

```
eval exec cc $CFLAGS -c $target
```

but beware of unwanted flattening in commands containing braces: Due to `eval` one level of braces disappears and

```
sed -e {s/^[/]*//} bla >bli
```

is flattened by `eval` to

```
exec sed -e s/^[/]*// bla >bli
```

resulting in complaints about the unknown command in `[/]`. The remedy is an extra level of braces.

1.12.5 A note on namespaces

As mentioned above, every `brasfile` is executed in its own autogenerated namespace. This shields `brasfiles` in different directories from each other. In particular, variables set and read without namespace qualification normally reference instances local to the namespace.

However, it is easy to forget that variables set globally, e.g.

```
set ::CFLAGS -g
```

can be seen in every namespace. This is **true also for subsequently setting the variable**. If, for example, the above is executed in one `brasfile` and another `brasfile` later calls

```
set CFLAGS -O2
```

the **global variable is changed** because it existed before. To explictly reference the namespace-local variable in cases where you suspect an already defined global, use

```
variable CFLAGS
set CFLAGS -O2
```

1.13 Compatibility with Older Versions

Up to version 0.8.x, `bras` had a different and less flexible concept of writing rules. The condition was implicitely given by the name of the command to register a rule. There were rule commands like `Newer`, `Exist` and `Always` as well as respective pattern rules. To be compatible with older versions, these commands are still maintained but are translated internally into calls to `Make`.

The translation is rather trivial and can be found at the end of the file `makeRule.tcl`. Since these commands are useful shortcuts in many common situations, they are listed below with their translation into the new scheme.

1.13.1 `Always`

`Always` *targets deps command*

is equivalent to

`Make` *targets* {`[true` *deps*`]`} *command*

1.13.2 `PatternAlways`

`PatternAlways` *regexp deptag command*

is equivalent to

`PatternMake` *regexp deptag* {`[true $d]`} *command*

1.13.3 `Exist`

`Exist` *targets command*

is equivalent to

`Make` *targets* {`[missing` *targets*`]`} *command*

1.13.4 `Newer`

`Newer` *targets deps command*

is equivalent to

`Make` *targets* {`[older` *targets deps*`]`} *command*

1.13.5 `PatternNewer`

`PatternNewer` *regexp deptag command*

is equivalent to

`PatternMake` *regexp deptag* {`[older` *target* `$d]`} *command*

1.14 `Bras` as a Package

Since Version 0.99.1 `bras` can be used as a package in any Tcl-script. In the future this will be better documented. In short, it should suffice to have

```
package require bras
namespace import ::bras::*
```

somewhere at the beginning of your script to be able to use `bras`' commands `Make`, `consider` and so on. Most command line options of `bras` can be set by calling `::bras::configure`.

If you use `bras` in a Tk-application, call the command `::bras::forget` (p. 41) before reconsidering targets. Otherwise `bras` knows that it considered them before and thinks there is nothing to do.

You may also want to redefine the procedure `::bras::report` (p. 44) to redirect all kinds of output into a widget.

In addition some care is necessary with regard to namespaces. If a `brasfile` is read implicitly by following an @-target or explicitly with `include` *@dir* for some directory *dir*, a namespace is set up for that directory. Now, whenever a target is considered in that directory, either by

1. `cd` *dir*
   `consider somefile`

   or by calling an @-target like

2. `consider @`*dir*`/somefile`

the namespace for that directory will be used. In particular, the rule's condition will be evaluated in that namespace as well as the rule's command.

The problem arises mainly if something like number (1) above is called **before and after** a `brasfile` for *dir* is read. Before, the namespace will be `::`, i.e. the global context, and afterwards it will be the namespace set up for that directory. See section 2.3.3 and 2.3.8 for more information on `dirns` and `linkvar` respetively.

1.15 Limitations and Known Bugs

Please report bugs to the author.

1. Since I did not yet test `bras` with a really big project like say `Xemacs` or `Tcl/Tk`, I don't know if the recursive inclusion of many `brasfiles` will lead to performance problems.
2. Parallel execution of several commands (as gnu make's option `-j`) is not yet supported.
3. A hack to translate `makefiles` to `brasfiles` is not available.
4. As normal in Tcl, parameters of `exec` are not passed through `glob`, which may lead to surprising error messages about non-existing files like `*.o`.
5. The rule-files containing default pattern-rules are not very elaborate.
6. Although Tcl is a portable platform, `bras` may still contain a few *nixisms.

1.16 The Name of the Game

First I wanted to call it `brassel` which is the imperative of `brasseln`, a verb you probably won't find in the dictionary. You may need to ask someone from around Köln,

but be prepared to get the answer that (s)he knows what it means but cannot explain it. I think the description "'working concentrated, busily but without stress"' is quite close to the real meaning.

Well, since `brassel` is much to long for a good Un*x utility, I shortened it to `bras`. It rhymes on the german word *Faß*.

1.17 Credits

Some people helped to push `bras` into the right direction. I would like to thank them.

- `nemo@gsyc.inf.uc3m.es` (Fco. J. Ballesteros) after looking at a very early version had the idea to allow the definition of new types of rules.
- Jason Gunthorpe <`jgg@debian.org`> started experiments with `bras` to get rid of the `make/automake`-combination with `bras`. The resulting long discussions resulted in several changes and enhancements of `bras`. In particular commands to fine-tune the rule-base were added, the semantics of pattern-rules was revised and the `DependsFile`-rule was born.
- Paul Duffin <`pduffin@hursley.ibm.com`> wanted to have the search path for dependencies.
- Paul D.Smith <`psmith@gnu.org`> explained some of the inner workings of `make` to me so that I was able to plagiate them.

2 Reference Guide

2.1 Predefined predicates

`Bras` comes with a set of predicates which cover the most common tasks and allow to mimic the function of `make`. If you develop a useful new predicate, send me the source.

2.1.1 dcold — is a target older than a dependency cache?

Synopsis:

`dcold` *target cache*

Description:

The predicate `dcold` checks if a file is out-of-date with respect to its dependency cache. A dependency cache is used to store all files an object file depends on, i.e. the respective source file as well as those source files which are (recursively) included by the first one.

The predicate expands `$cache` along the search path and then passes it to the reasoning process to make sure the file is up-to-date. It then checks with the predicate `older` if the `target` is older than any of the files listed in the file `$cache`.

Example:

See file `c2o.rule` for an example of how to use `dcold`.

2.1.2 md5older — did the md5sum of a dependency change?

Synopsis:

`md5older` *target deps*

Description:

The predicate `md5older` maintains for the given target a cache file which contains the md5sums of the given dependencies. It returns true if either the cache file does not exist or if at least one of the dependencies has an md5sum not listed or different from the one in the cache file. These dependencies will be communicated to the rule's command in variable `trigger`. If the cache file does not exist, all dependencies will show up in `trigger`. The list of all dependencies can be picked up in `deps`.

The name of the cache file is *target*`.md5`.

Note: Unlike predicate `older`, `md5older` does not test whether the target does exist as a file. Consequently, if you run `bras` and then delete the target, it will not be rebuilt automatically if you only use `md5older`. Instead use a condition like

```
[or [md5older ...] [missing ...]]
```

2.1.3 missing — is a file missing?

Synopsis:

`missing` *name*

Description:

The predicate `missing` returns *true* if and only if the given file does not exist. In that case, its name will be appended to the variable `trigger` and can be found therein by the command associated with rule to which the predicate belongs.

Example:

```
Make /usr/local/lib/bla {[missing $target]} {
  file mkdir $target
}
```

2.1.4 notinitialized — was a target not yet called?

Synopsis:

`notinitialized` *targets*

Description:

This predicate maintains an internal list of those targets for which it was called before. If one of its arguments is not yet on the list, it returns `true`.

Please note that `notinitialized` **does not** call `consider` for its arguments.

Example:

```
Make {paramA paramB paramC} {[notinitialized $targets]} {
  source paramfile
  # assumes that variables paramA ... are set
}
```

This predicate is mainly useful in applications with a GUI which make use of `forget` (p. 41) because otherwise a target is never considered more than once. As shown above, it can be used to initialize variables from a source-file once.

2.1.5 oldcache — is a dependency cache out-of-date?

Synopsis:

`oldcache` *cache client*

Description:

The predicate `oldcache` checks if the given dependency cache (`$cache`) is out-of-date.

A dependency cache is typically used to store all the files an object file depends on, in particular the respective source file, i.e. its client, as well as any other source files which are (recursively) included by the client.

The dependency cache is out-of-date, if it either does not exist, if it is older than the client-file, or if it is older than any of the files listed in itself. Here *older* is meant in the sense of predicate `older`.

This predicate sets the variables `trigger` and `deps` for the command of the rule to which it belongs. The client file will be listed before any files it includes.

Example:

See file `cdeps.rule` for an example of how to use `oldcache`.

2.1.6 older — are targets older than dependencies?

Synopsis:

`older` *targets deps*

Description:

The predicate `older` returns *true* to the reasoning process, if any of the names listed in its first parameter either does not exist or is older than any of the files listed in its second argument. Before `older` performs its test, it expands all elements in `deps` along the search path and passes them to the reasoning process to make sure they are up-to-date. If after considering a dependency there is still no file with that name, `older` uses the result returned by the reasoning process to decide if this dependency was just made. If the result is *true*, the dependency is considered *very new*, otherwise it is considered *very old*. If however the dependency exists as a file, its modification time is used.

2.1.7 pairedolder — is i'th target older than i'th dependency?

Synopsis:

`pairedolder` *targets deps*

Description:

For every element of the list *targets*, the predicate checks if it does not exist as a file or is older than the respective element in *deps*. Contrast this with the predicate `older` which checks **every** element of `targets` against **every** element of *deps*.

All dependencies are considered before the comparison starts. If a dependency is nevertheless not available as a file, the report returned by the reasoning process replaces the file modification time: if the dependency was made, it is treated as being newer than the respective target, otherwise it is supposed to be older.

Example:

The main reason to introduce this predicate was to support Java compilation.

> **The Java case**
> As of today (Feb. 2002) the typical Java Development Kit (jdk) does not help in generating decent `make`- or rather `bras`-dependency information. But even if `jikes`' option `+M` comes to the rescue, there is the problem that class files can be mutually dependent. The resulting dependency loop can not be handled by `bras`.
>
> Anyway, in newsgroups it seems to be agreed upon that it is best to compile all source files in one call to the Java compiler. This is what is supported by `pairedolder` as demonstrated below.

To compile all Java source files found in a directory, use:

```
set JAVAC javac
set JFLAGS ""
set JAVA [glob *.java]
regsub -all {[.]java} $JAVA .class CLASSES
Make $CLASSES {[pairedolder $CLASSES $JAVA]} {
  $JAVAC $JFLAGS $deps
}
```

When `bras` is asked to consider any one of the class files, it will compile all source files, but only if any of them is newer than the respective class file.

It is convenient to add the following shortcut target to the `brasfile`:

```
Make classes {[updated $CLASSES 0]} .relax.
```

It allows to just call

```
bras classes
```

on the command line to compile all Java source files.

2.1.8 true — always returns true

Synopsis:

`true` *deps*

Description:

The predicate `true` expands all its arguments along the search path and passes them to the reasoning process to make sure they are up-to-date. The are also passed to the command of the rule to which the predicate belongs in the variable `deps`.

Note:

This predicate breaks the chain of reasoning because it always returns 1 (`true`). Use of `updated` is recommended if this is not intended.

2.1.9 updated — does a target need an update?

Synopsis:

`updated` *deps*

Description:

The reasoning process is called for all given dependencies. If any of them is updated during that process, `updated` returns `true`.

This predicate is normally used to check the dependencies of targets which themselves are no file targets. Compare this to the predicate `older` (p. 2.1.6) which yields `true` if the target does not exist, even if none of the dependencies is out-of-date. In contrast, normally you don't even pass the target to `updated`.

Example:

The following rule combines the update decision about the three libraries into one target, namely `libs`, which is then used as a shortcut representing the three libraries.

```
Make libs {[updated {libR.a libS.a libT.a}]} {
  # Nothing to be done here, libs is not a file target.
  # It only serves as a shortcut to combine in an
  # OR-like fashion the results of considering the
  # dependencies.
}
Make program {[older $target {... libs ...}]} {
  #compilation or other things
}
```

2.1.10 varchanged — did a variable's value change?

Synopsis:

`varchanged` *varnames cachefile*

Description:

The predicate `varchanged` tests if any of the given variables is different from their value as found in *cachefile*. The list *varnames* may contain names of scalar variables, array elements or arrays, however in almost all cases those names must be fully namespace qualified, i.e. they must start with `::`.

The test proceeds as follows:

- If the *cachefile* does not exist, the predicate returns *true*. Otherwise the file is `sourced` in, i.e. it must contain Tcl code.

- If a name mentioned in *varnames* is not found in *cachefile*, the predicate returns *true*.
- If the value — or values, in case of an array name — referenced by a name in *varnames* differs from what is found in *cachefile*, the predicate returns *true*.
- Otherwise, the predicate returns *false*.

All names in *varnames* are considered as targets **before** the test is run. Those names the values of which were found to differ from those in *cachefile* will show up in the automatic variable `trigger`.

Example:

Suppose a parameter file is used to store parameters of algorithms, and you want an algorithm to be run only if a parameter relevant to that algorithm changes. Consequently the result produced by an algorithm shall not depend on the parameter file as a whole, but only on that particular parameter.

The following code shows an example with two result files, two input files and one parameter file:

```
##############################################################
# $Revision: 1.2 $, $Date: 2000/06/22 11:33:00 $
##############################################################
Always all {resultA resultB} {
}

Make resultB {
  [or [older $target inputB] [varchanged ::paramB $target.cache]]
} {
  echo $paramB >$target; # a fake computation
  echo "set paramB $paramB" >$target.cache
}

Exist inputB {
  touch $target
}


Make resultA {
  [or [older $target inputA] [varchanged ::paramA $target.cache]]
} {
  echo $paramA >resultA; # a fake computation
  echo "set paramA $paramA" >$target.cache
}

Exist inputA {
  touch $target
}

Make {::paramA ::paramB} {[notinitialized $targets]} {
```

```
  include params
}
#################################################################
```

When `bras` is called to make `resultA` it acts according to rule

```
Make resultA ...
```

In particular it checks the predicate

```
[varchanged ::paramA $target.cache]
```

to see if variable `::paramA` has changed according to the given cache file. Predicate `varchanged` first considers `::paramA` as a target, which allows the last rule to initialize it from a parameter file (see page 28 for `notinitialized`). After that, the cache file, if available, is read and the current value of `::paramA` is compared to what is found in the cache file. If the cache file does not exist, if it does not set `::paramA` or if the current value is different from the stored one, `varchanged` returnes *true*. As a consequence, the associated script is run and updates the files `resultA` and `resultA.cache`.

Note that the cache file should normally be written at the same time as the computation is performed to keep it up-to-date. Also note that the cache file is of course **not** the same file as the parameter file.

2.2 Predefined Pattern Rules

The distribution contains a small collection of pattern rules. To use them you have to include them explicitly in your rule file. For example to load the rule to install files, use

```
include [file join $::bras::base install.rule]
```

The following sections describe most available rules. Check the distribution for `.rule`-files for recent additions not covered by the documentation below.

2.2.1 c2o — compile C source into object file

Synopsis:

```
include [file join $::bras::base cdeps.rule]
include [file join $::bras::base c2o.rule]
```

Description:

Compiles C source files into object files.

Target:

```
{.*\[.]o(bj)?}
```

Dependency:

The dependency is derived from the target by replacing the target's suffix with `.c`.

Predicate:

The target is checked with `dcold` (see 2.1.1) against its dependency cache file maintained by `dcold`.

Command:

```
$CC -o $target -c $CFLAGS [lindex $deps 0]
```

For C++ use `c++2o.rule` instead.

2.2.2 c++2o — compile C++ source files

Synopsis:

```
include [file join $::bras::base c++deps.rule]
include [file join $::bras::base c++2o.rule]
```

Description:

Compiles C++ source files into object files.

Target:

```
{.*\[.]o(bj)?}
```

Dependency:

The file `c++2o.rule` actually contains 4 pattern rules which relate the target to a dependency with one of the suffixes `.C`, `.cc`, `.cxx` or `.cpp`.

Predicate:

The target is checked with `dcold` (see 2.1.1) against its dependency cache file maintained by `dcold`.

Command:

```
$CXX -o $target -c $CXXFLAGS [lindex $deps 0]
```

2.2.3 cdeps — maintain dependency cache for C source files

Synopsis:

```
include [file join $::bras::base cdeps.rule]
```

Description:

Maintains a dependency cache for a source file.

Target:

```
{.*\[.]dc}
```

Dependency:

The dependency is derived from the target by replacing the target's suffix with `.c`.

Predicate:

The target is checked with `oldcache` (see 2.1.5) against the dependency.

Command:

```
::bras::updateCacheC \
   $target [lindex $deps 0] CC DEPOPTS CDEPEXCLUDE
```

Set `DEPOPTS` to the options which instruct your C compiler to emit dependency information for `make`. Optionally set the variable `CDEPEXCLUDE` to a regular expression for dependencies which need not be in the dependency list. A good candidate on *NIX systems is `^/usr/.*`.

For `C++` use `c++deps` instead.

2.2.4 c++deps — maintain dependency cache for C++ source files

Synopsis:

```
include [file join $::bras::base c++deps.rule]
```

Description:

Maintains a dependency cache for a source file.

Target:

```
{.*\[.]dc}
```

Dependency:

The dependency is derived from the target by replacing the target's suffix with one of the suffixes `.C`, `.cc`, `.cxx` or `.cpp`.

Predicate:

The target is checked with `oldcache` (see 2.1.5) against the dependency.

Command:

```
::bras::updateCacheC \
   $target [lindex $deps 0] CXX CXXDEPOPTS CXXDEPEXCLUDE
```

Set `CXXDEPOPTS` to the options which instruct your `C++` compiler to emit dependency information for `make`. Optionally set the variable `CXXDEPEXCLUDE` to a regular expression for dependencies which need not be in the dependency list. A good candidate on *NIX systems is `^/usr/.*`.

2.2.5 cli2ch — create command line interface with `clig`

Synopsis:

```
include [file join $::bras::base cli2ch.rule]
```

Description:

Uses `clig`[Kirsch 2000] to create C source and header files for a command line interpreter based on a description in a `.cli` file.

Target:

```
{.*\.[ch]}
```

Dependency:

The dependency is derived from the target by replacing the target's suffix with `.cli`.

Predicate:

The target is checked against the dependency by means of the `older` predicate (see 2.1.6).

Command:

The program `clig` is run to create the C source and header file.

2.2.6 install — install a file

Synopsis:

```
include [file join $::bras::base install.rule]
```

Description:

The rule creates a relation between a file to be installed somewhere in the system and a file in the local directory or search path (see 1.8). If the local file is older than the target, the function `::bras::install` (see section 2.3.7) is used to install the file. To identify a target as a file to be installed, it must look like

```
/some/installation/dir/theFile/0755
```

On UNIX, the trailing octal number will be used to set the access permissions of the installed file. On other platforms, it is simply ignored.

In addition to the behaviour described above, `bras` also looks for a local file `theFile.fixed`. Sometimes a path must be edited into a script or documentation file just before it is copied to its final destination. It is expected that the late fix results in a local file with suffix `.fixed`.

Target:

```
[file join .* .* {0[0-9][0-9][0-9]}]
```

Dependency:

The dependency is generated by stripping from the target the trailing permissions as well as any directories. Before the resulting plain dependency, the dependency with suffix `.fixed` is tried.

Predicate:

The `older` predicate (see section 2.1.6) is used to check if the target to be installed is out of date with respect to the local file.

Command:

The target with permissions removed, the dependency found and then the permissions are passed to `::bras::install` (see section 2.3.7).

2.2.7 lex2c — (f)lex source to C source

Synopsis:

```
include [file join $::bras::base lex2c.rule]
```

Description:

Relates a C source file `*.c` to a (f)lex source file `*.lex`. In particular, (f)lex is called with option `-t` to make sure that the output file is not called `lex.yy.c`.

Target:

```
{.*[.]c}
```

Dependency:

The suffix `.c` replaced by `.lex`.

Predicate:

`older` (see 2.1.6)

Command:

```
    $LEX $LEXFLAGS -t [lindex $deps 0] >$target
```

2.2.8 o2x — *IXish linking of object files into executable

Synopsis:

```
include [file join $::bras::base o2x.rule]
```

Description:

Maintains an executable in relation to an object file. Since an executable usually depends on several object files, this pattern rule is mainly of use to implicitly define a command to link the files rather than deriving the dependency relation automatically (see the discussion in section 1.5.2). The dependency relation with all the object files must usually be described by other means.

Target:

A name which does not contain a dot, i.e. `{[^.]*}`

Dependency:

The dependency is derived from the target by suffixing it with `.o`.

Predicate:

The target is checked against the dependency by means of the `older` predicate (see 2.1.6).

Command:

```
$CC -o $target $CFLAGS $LDFLAGS $deps $LDLIBS
```

2.3 New Commands

The commands described in this section are all declared in the namespace `::bras`. They are all automatically imported into the global namespace when `bras` starts. However, when using `bras` as a package, they do not appear automatically in the global namespace and must be imported with

```
namespace import ::bras::*
```

after the call to `package require`.

2.3.1 .relax. — a do-nothing indicator for rules

Synopsis:
`.relax.`

Description:
Used in place of a rule's command, `.relax.` indicates explicitly that there is nothing to do for the target. Contrast this with using

1. the empty string. It means that there is no command at all and will eventually trigger a warning messages, if no other rule for the target is available wich has a command.
2. a string of white space. If the target is rendered out-of-date, it is actually executed, does of course nothing, but emits a message like

```
# making target
```

To suppress the message, use `.relax.` instead of the string of white space.

Note:
Although it looks like a Tcl command, `.relax.` is none. `Bras` does not evaluate it. It only applies a string compare to a rule's command to find out if it is equal to `.relax.`. As an added convenience, white space space is trimmed before comparison.

Example:
The indicator `.relax.` is most conveniently used for default targets which only delegate work to a bunch of other targets but don't themselves represent files to be made.

```
Make all {[updated [progs docs]]} {
  .relax.
}
```

2.3.2 consider — explicitly call the reasoning process

Synopsis:
`consider` *targets*

Description:
The command calls the reasoning process explicitly for the given targets. In most cases, the reasoning process is called explicitly for a given target by specifying it on the `bras`-commandline. All other invocations happen automatically by recursively considering dependencies of targets under consideration. However, just in case the need arises, here is the command to invoke the reasoning process.

Example:

A truly useful example were the need arises to call `consider` is the maintenance of dependency caches as shown in the distribution file `cdeps.rule`.

2.3.3 dirns — find namespace associated with directory

Synopsis:

`dirns` *dir*

Description:

Retrieves the name of the namespace associated with the `brasfile` in directory *dir*. The association is created only when a `brasfile` in *dir* is sourced either implicitly by following an `@`-target or explicitly by `include @`*`dir`*. If no such association was created before, the global namespace `::` is returned.

Example:

```
# Use a configuration option defined in the brasfile
# of the parent directory to change behaviour.
if {[set [dirns ..]::XYZOPTION]} {
  ...
} else {
  ...
}
```

2.3.4 forget — forget which targets where already considered

Synopsis:

`forget` ?*targets dirs*?

Description:

Both parameters *targets* and *dirs* can be glob-patterns. They describe which targets `bras` shall mark as “not yet considered’ in which directories. If *dirs* is not given, `bras` forgets the matching targets in all directories. If no argument is given, `bras` forgets all targets.

The functions is normally not useful if `bras` is invoked on the command line. However if the `bras`-package is used in a Tk-application, `forget` allows to reconsider targets after the user changed some settings.

2.3.5 getenv — set variable from environment of from default

Synopsis:

`getenv` *name* ?*default*?

Description:

The command copies the element of `env` with the given name into a variable of the same name. If the optional default value is not given and an element of the given name does not exist in `env`, an error message is printed and `bras` exits. If a default value is given, it is taken instead of the missing entry in `env`.

Example:

A typical use of `getenv` is to set compiler switches like `CFLAGS`:

```
getenv CFLAGS {-W -Wall -g}
```

Please note that `bras` allows to set elements of `env` on the command line (see section 1.12.1), so the above default can be overridden by calling `bras` like

```
/home/bobo> bras CFLAGS='-O2 -W -Wall'
```

2.3.6 include — source file only once

Synopsis:

`include` *@dir*
`include` *file*

Description:

Like `source`, `include` reads and executes a Tcl-file. In contrast to `source`, `include` makes sure to source every file only once.

If the parameter is the name of a directory prefixed with `@`, the `brasfile` of the given directory is sourced in the same way as if `bras` had followed an `@`-target to that directory. In particular, the file is sourced within its own namespace.

The name of the `brasfile` used depends on the name of the file where this command is found in. If `bras` was called with option `-f` specifying a special name, that name is also used in the destination directory.

If the parameter given does not start with `@`, the file is sourced in the global namespace.

Example:

The form

```
include @..
```

is most often used in `brasfile`s of subdirectories to include a standard rule file of the parent directory. It allows to call `bras` in the subdirectory alone and still having all global definitions of `../brasfile`.

2.3.7 install — install a file

Synopsis:

`install` *target source* {*perm* {}}

Description:

Installs a file *source* as a file *target*. Before file *source* is copied onto the *target*, the targets directory is made with with the command `[file mkdir]`. On UNIX, the target's access permissions are set as requested by *perm*. On other platforms, *perm* is ignored. This command is used by the pattern rule defined in `install.rule` (see 2.2.6).

2.3.8 linkvar — link variable to other namespace

Synopsis:

`linkvar` *varname ?varname? ... dir*

Description:

Make the variables given into aliases for variables of the same name in the namespace associated with the `brasfile` in directory *dir*. For this to work, the `brasfile` in *dir* must have been sourced already either implicitly by following an @-target or explicitly by `include @dir`.

Example:

```
# use same CFLAGS as in parent directory
linkvar CFLAGS ..
```

2.3.9 report — callback for warnings and error messages

Synopsis:

`report` *type text* ?*newline*?

Description:

`Bras` uses this function to produce all kinds of output on the console. If you want this output to go anywhere else, redefine this procedure such that it matches the interface described here.

The possible values of parameter `type` and their meaning is described in the following table.

parameter	printed on	meaning
`warn`	`stderr`	warning messages
`-v`	`stdout`	output triggered by option -v
`-ve`	`stdout`	output triggered by option -ve
`-d`	`stdout`	output triggered by option -d
`norm`	`stdout`	normal output without options

The default implementation of `report` prints with `puts` and suppresses the trailing newline only, if parameter `newline` is explicitly set to 0.

2.3.10 searchpath — set or get search path

Synopsis:

`searchpath` ?*new_path*?

Description:

Without an argument, the search path set for the current directory is returned. If none is set, the empty string is returned. If an argument is given, it is the list of directories where `bras` searches for dependencies (see section 1.8).

Note that the searchpath is local to a given directory, i.e. if `bras` follows an @-target to another directory, that directory has its own search path.

Example:

You may want to make the search path dependent on the platform on which you are building.

```
searchpath [list . ../generic ../$tcl_platform(platform)]
```

2.4 Miscellaneous Commands

The commands described in this section are (hopefully) of general interest when using `bras`. They are all defined in namespace `::bras` and are not automatically imported. To use them, either call them with their fully qualified name, e.g. `::bras::packjar`, or import them with `namespace import`.

2.4.1 cvsknown — return all files CVS knows about

Synopsis:

`::bras::cvsknown`

Description:

The procedure recursively finds the files `CVS/Entries` and returns the catenated list of file names it finds in them. Consequently, the result is a list of all files CVS knows about in the current directory and its subdirectories.

Example:

The procedure can be used to keep an archive file of a CVS-module up-to-date with respect of the file in the module.

```
set module [file tail [pwd]]
Make ${module}.tar.gz {
  [older $target [::bras::cvsknown]]
} {
  ## export module from CVS and pack it
}
```

Only if `$module.tar.gz` is actually considered for update the rather expensive procedure `::bras::cvsknown` is run. When other targets are considered, it is never executed.

2.4.2 makedeps2bras — convert `make` dependencies

Synopsis:

`::bras::makedeps2bras` *text exclude*

Description:

The command assumes that *text* contains `make` dependencies like they are automatically generated by some compilers, e.g. with `gcc -M` or `jikes +M`. In particular the text should contain lines like

```
target:  dep1 dep2
target:  dep3 dep4
```

Continuation lines are allowed. The command fetches **only** the dependencies, assuming that all targets mentioned are the same. If *exclude* is not the empty string, it should be a regular expression. Dependencies which match this expression will not be included in the resulting list. An example would be `^/usr/.*`.

Example:

Take a look at the file `updateCacheC.tcl` of the distribution to see how this command is used.

2.4.3 packjar — create a java archive from java packages

Synopsis:

`::bras::packjar` *jar pkgroot pkgdirs* ?*glob*?

Description:

The command creates a java archive in the file *jar*. It packs files matching the pattern specified in *glob* in any of the package directories listed in *pkgdirs*. The default pattern is `*.class`. Package directories are interpreted relative to *pkgroot*.

Example:

To pack all class files as well as all PNG-images of the package `text.regexp`, use

```
set PKGROOT [file dir [file dir [pwd]]]
set PKG [file join [split text.regexp .]]
::bras::packjar Regexp.jar $PKGROOT \
                    $PKG [list *.class *.png]
```

Bibliography

[Miller 1997] P. Miller: *Recursive Make Considered Harmful.* http://www.pcug.org.au/~millerp/rmch/recu-make-cons-harm.html is included in the `bras`-distribution as `recu-make-cons-harm.ps.gz`

[Kirsch 2000] H. Kirsch: *Command Line Interpreter Generator.* http://wsd.iitb.fhg.de/~kir/clighome/

Index

U

V