Specware 4.0 User Manual

Specware 4.0 User Manual

Table of Contents

1. Installing Specware....4

1.1. System Requirements....4

1.1.1. Hardware....4

1.1.2. Operating system....4

1.2. Installation Instructions....4

2. Getting Started....6

2.1. Starting Specware....6

2.2. Exiting Specware....6

3. Usage Model....7

3.1. Units....7

3.2. Interaction....7

4. Processing Units....9

4.1. Overview....9

5. Debugging Generated Lisp Files....10

5.1. Section 1: Tracing....10

5.2. Section 2: Breaking....10

5.3. Section 3: Timing....11

5.4. Section 4: Profiling....12

5.5. Section 5: Interrupting....13

Chapter 1. Installing Specware

1.1. System Requirements

1.1.1. Hardware

Specware has relatively modest system requirements for simple projects. Of course, as with any development tool, as your projects being developed become more complex, you may wish to work on a more powerful machine. For average use, however, the following basic hardware configuration is recommended:

- CPU: 250 Mhz
- RAM: 128 MB total, at least 64 MB free for applications
- Disk space: 15 MB for base system, 10-50 MB for user projects

1.1.2. Operating system

Specware 4.0 has been tested to work with Windows NT 4.0 and Windows 2000.

1.2. Installation Instructions

1. Before running the Specware 4.0 installer, you should have a recent version of XEmacs (21.1 or higher) installed on your machine. If you already have XEmacs, then proceed to the next step. Otherwise, open the XEmacs folder provided on the setup CD and run `setup.exe`. Select "Install from Local Directory" as the Installation Method, and click "Next" through the remaining steps to accept the

default configuration. Note: If you choose not to complete this step of installing XEmacs, a text-only version of Specware will be installed instead.

a. After XEmacs has been installed, launch the Specware 4.0 installer `setup.exe` on the CD.

b. Follow the instructions in the installation wizard, and Specware 4.0 will be installed on your machine. A shortcut to Specware will be placed on your Desktop as well as in the Program Files folder in the Start menu.

Chapter 2. Getting Started

Specware is a development environment that runs on top of Lisp. This chapter describes the Specware environment and the basic mechanisms for running Specware.

2.1. Starting Specware

To start Specware, double-click the `Specware 4.0` shortcut on your Desktop, or select `Specware 4.0` from the `Start Menu -> Program Files -> Specware` folder. When Specware is launched, a couple things happen: XEmacs is started and a Lisp image containing Specware is started inside an XEmacs buffer. All of the user interaction (see the next chapter) with Specware occurs at the Lisp prompt.

2.2. Exiting Specware

To exit Specware, type `:exit` at the Lisp prompt. A message will appear indicating that another process exists. Type `y` to confirm that you want to exit. This will kill Specware and you may then close the XEmacs window.

Chapter 3. Usage Model

3.1. Units

Simply put, the functionality provided by Specware consists in the capability to construct specs, morphisms, diagrams, code, proofs, and other entities. All these entities are collectively called *units*.

Some of the operations made available by Specware to construct units are fairly sophisticated. Examples are colimits, extraction of proof obligations, discharging of proof obligations by means of external theorem provers, and code generation.

The Metaslang language is the vehicle to construct units. The language has syntax to express all the unit-constructing operations that Specware provides. The user defines units in Metaslang, writing the definitions in `.sw` files (this file extension comes from the first and fifth letter of “Specware”").

Currently, the only way to construct units in Specware is by writing text in Metaslang. Future versions of Specware will include the ability to construct units by other means. For instance, instead of listing the nodes and edges of a diagram in text, it will be possible to draw the diagram on the screen.

3.2. Interaction

All the interaction between the user and Specware takes place through the Lisp shell. The `.sw` files that define units are edited outside of Specware, i.e. using XEmacs, Notepad or any other text editor of choice. These files are processed by Specware by giving suitable commands at the Lisp shell.

When `.sw` files are processed by Specware, progress and error messages are displayed in the XEmacs buffer containing the Lisp shell. In addition, the results of processing are saved into an internal cache that Specware maintains. Lastly, processing of certain kinds of units result in new files being created. For example, when Lisp code is generated from a spec, the code is deposited into a `.lisp` file.

Specware also features auxiliary commands to display information about units, inspect and clear the internal cache, and inspect and change an environment variable that determines how unit names are resolved to `.sw` files.

Chapter 4. Processing Units

4.1. Overview

Unit definitions are processed by Specware. The user instructs Specware to process units by supplying certain commands. Specware has access, via the Lisp run time environment, to the underlying file system, so it can access the unit definitions contained in `.sw` files. The environment variable [[SWPATH]] determines which `.sw` files are accessed by Specware to find unit definitions.

Processing of a certain unit causes processing of the units referenced in the defining text of the unit, recursively. For instance, if a spec `A` extends a spec `B` which in turns extends a spec `C`, when `A` is processed also `B` and `C` are processed. There must be no circularities in the chain of unit dependencies.

Processing causes progress and/or error messages to be displayed on the screen, in the XEmacs buffer containing the Lisp shell (where the user also supplies commands to Specware). Progress messages inform the user that units are processed without error. Error messages provide information on the cause of errors, so that the user can fix them by editing the unit definitions.

The processing of certain kinds of units results in the creation of new files, as an additional side effect. For instance, Lisp programs are a kind of unit, constructed by the `generate` operator of Metaslang. A side effect of processing one such unit is that the resulting code is written into a `.lisp` file.

Chapter 5. Debugging Generated Lisp Files

5.1. Section 1: Tracing

If you need to debug(!) your application, there a number of useful lisp facilities you should be aware of. The simplest trick is to trace some functions you care about to see what they are doing.

```
(trace foo)
  This will display the arguments to foo each time it is
  called, and will display the results each time it returns.

(untrace foo)
  This will turn off any tracing on foo.
```

5.2. Section 2: Breaking

If you need a more detail view of runtime behavior, you might want to BREAK some functions you care about.

```
(trace (foo :break-all t))

  This will invoke the debugger each time foo is called,
  and upon each exit from foo.
```

Once you arrive in the debugger, the following commands are most useful:

```
 :down <n>    Move to a deeper frame  <n> is optional
 :up <n>      Move to a higher frame  <n> is optional
 :zoom <n>    Display n frames        <n> is optional

 :pri         Enter a dialog that lets you set printer control
```

```
              variables.  For example, set-
ting depth to 5 and length
              to 10 will let you see the top level structure of
              expressions, while suppressing deep expres-
sions and
              the tails of long expressions.
              Note that you'll need to specify values for many
              contexts, but just hitting return leaves a value
              unchanged.  You'll likely want to mod-
ify the trace,
              debugger, and current values.

 (pprint *)   Pretty print the expression for the current frame.
              Note that this only works immediately af-
ter arriving
              at a frame, e.g. via :down 0 if necessary.

 :cont        Continue as if nothing happened.
 :restart     Resume execution at this frame.
 :reset       Re-
turn to lisp top level.  (E.g., bail out to try again.)
 :exit        Exit from lisp to operating system - ends session.

 :help        Online documentation.

 (misc ...)   The debugger is is a read/eval/print loop, so
              arbitrary lisp forms will be evaluated (in the
              current dynamic context).

 (untrace foo)  Stop entering debugger when foo is called.
                Note that you may still enter the debugger for
                each exit from calls to foo already recursively
                in progress.
```

5.3. Section 3: Timing

If you are curious about the overall performance of your application, the TIME macro will provide some quick information:

```
(time (foo nil))
  This will report the time and space used by foo, e.g.:

 USER(1): (time (list 1 2 3))
 ; cpu time (non-gc) 0 msec user, 0 msec system
 ; cpu time (gc)     0 msec user, 0 msec system
 ; cpu time (total)  0 msec user, 0 msec system
 ; real time  231 msec
 ; space allocation:
 ;  6 cons cells, 0 symbols, 0 other bytes, 0 static bytes
 (1 2 3)
```

Note that TIME it transparent, i.e., it returns whatever its argument would return, including multiple values, etc., so it is safe to intersperse it nearly anywhere.

Common Lisp has more facilities for rolling your own timers: see the generic Common Lisp documentation, or contact Kestrel Technologies.

5.4. Section 4: Profiling

The following code, specific to Allegro Common Lisp, will produce a detailed analysis of the time and space used by each function invoked. The time usage is done by polling at 20 msec intervals, so this is not particularly useful for very short-lived calls. See the Allegro documentation for many variations on this theme.

```
(progn
 (prof::with-profiling (:verbose t) (foo nil))
 (prof::show-flat-profile)
 (prof::show-call-graph)
 )
```

Note that the report for the call graph may look confusing at first glance, but after a small learning effort, it is actually quite informative and easy to read once you understand the format. If you have any questions, contact Kestrel Technologies.

5.5. Section 5: Interrupting

Finally, note that a useful trick in lisp is to start your application, e.g. (foo nil), then at an appropriate time hit control-C. This will interrupt your application and put your into the debugger. From there you can enter the command :zoom to see the top of the stack. That can often be quite revealing.

Again, see the Allegro documentation for voluminous details, or contact Kestrel Technologies.