NAME

sr − SR compiler

SYNOPSIS

sr [−sbcqwegvOTMCFP] [−I dir **] [−o** file **] [−d** flags **]** file.sr ... otherfiles...

DESCRIPTION

Sr compiles programs written in the SR language. In the usual case, one or more **.sr** files are compiled to produce **.o** files, and then the SR linker, *srl*(1), is called to produce an executable file **a.out**. The last encountered resource is taken as the main resource.

Compilation and linking takes place in the context of an *Interfaces* directory used for passing specification and object files between *sr* and *srl* runs. Files are created in the Interfaces directory as each resource or global is compiled. The Interfaces directory is needed for compilation and linking, but not execution, and can be deleted after an executable has been produced.

Each source file is read twice. The first pass copies component specifications into the Interfaces directory; the second pass generates object code. If errors are detected in the first pass, the second pass is omitted.

Files not ending in **.sr** (such as **.o**, **.a**, and **−l***xx* files) are not processed by *sr* but are passed along to *srl*.

OPTIONS

Any of the options **s**, **b**, **c**, **M**, or **C** inhibits linking.

−I *dir* Use *dir*/Interfaces as the Interfaces directory instead of ./Interfaces.

−o *file* Use *file* for the executable instead of **a.out**.

−s Create specification files only (first pass only).

−b Compile bodies only (second pass only), leaving specification files unaltered.

−c Compile specs and bodies, but do not link **.o** files to create an executable file.

−q Suppress the echoing of source file names as they are compiled.

−w Suppress warning messages.

−O Omit some runtime error checks and invoke the C optimizer to improve the generated code.

−T Omit the ‘‘timeslicing’’ code that allows context switching at the top of each loop.

−M Generate dependency information for use by *srm*(1), then exit without compiling anything.

Additional options are useful mainly when debugging the compiler:

−C Stop after generating **.c** files, leaving them undeleted.

−F Inhibit constant folding. This can break programs by rendering certain expressions non-constant.

−P Inhibit normal optimizations in the SR compiler.

−d *flags* Write debugging output selected by *flags* on standard output.

−e Use experimental versions of *srl* and **.h** files, and pass **−e** to *srl.*

−g Compile for debugging with *dbx*(1), don’t delete **.c** files, and pass **−g** to *srl.*

−v Announce version number, trace other programs invoked by *sr*, and pass **−v** to *srl*.

ENVIRONMENT

SR_PATH

When compiling an **import** statement, *sr* searches for a corresponding **.spec** file by looking first in the current directory, then in the Interfaces directory, and finally in the SR library. If SR_PATH is set, any directories named there are checked ahead of the standard search sequence. Directories in SR_PATH are separated by colons (**:**).

EXECUTION ENVIRONMENT

These environment variables can be set at execution time to affect the behavior of an SR program:

SR_PARALLEL
: Controls the number of processes that can run with true concurrency (as opposed to simulated concurrency) under MultiSR. MultiSR is a configuration option available on the SGI Iris, Sun Sparc running Solaris 2.x, and Sequent Symmetry. The default value is 1 (no true concurrency). It makes little sense to set SR_PARALLEL greater than the number of available processors.

SR_SPIN_COUNT
: In MultiSR, the number of times an idle processor will check for a task before relinquishing the CPU to the operating system. Larger values increase the opportunity for processes to quickly resynchronize at the expense of greater CPU usage. The default is 35.

SR_NAP_INTERVAL
: In MultiSR, the number of milliseconds an idle processor will nap after trying unsuccessfully to find something to run. The default is 10.

SR_TRACE
: If not null, SR_TRACE specifies the name of a file to receive a trace of messages, invocations, and other language-level events. Two special names, *stdout* and *stderr*, direct the trace output to standard output and standard error output respectively. The default is no tracing. Traces from remote virtual machines appear only if the trace is directed to *stdout* or *stderr*. The *srprof*(1) program can be used to summarize the trace output.

SR_DEBUG
: A hexadecimal number specifying a bit mask that enables debugging information of various internal runtime events on standard error output. Each bit selects a different message category, and the higher order bits (FFFFFF00) produce the most voluminous output. Details are contained in the source code of the runtime system. The default is no debugging. In contrast with SR_TRACE, the output from SR_DEBUG records internal events of interest to maintainers of the runtime system.

SRXPATH
: Path to the executable of *srx*, which serves as the central controller of a distributed SR program. The default is configured when the SR system is built.

SRMAP
: File to read for the network configuration information used to generate a host-independent ‘‘network path’’ of the executable for executing part of an SR program on a remote host. The default is configured when the SR system is built.

FILES

file.sr	SR source file
a.out	executable program
Interfaces/component.c	C language intermediate file
Interfaces/component.spec	specification file
Interfaces/global.impl	implementation characteristics of global
Interfaces/component.o	object file

SEE ALSO

Gregory R. Andrews and Ronald A. Olsson, *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, 1993, ISBN 0-8053-0088-0.

Gregory R. Andrews, *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991, ISBN 0-8053-0086-4.

Ronald A. Olsson, Gregory R. Andrews, Michael H. Coffin, and Gregg M. Townsend, *SR: A Language for Parallel and Distributed Programming.* TR 92-09, Dept. of Computer Science, The University of Arizona, 1992. Included in the SR distribution.

Gregory R. Andrews, Ronald A. Olsson, et al., *An Overview of the SR Language and Implementation.* ACM Trans. on Prog. Lang. and Systems 10, 1 (January, 1988), 51-86.

srl(1), srm(1), srprof(1), cc(1), dbx(1), srmap(5), srtrace(5)

DIAGNOSTICS

Messages diagnosing erroneous programs are intended to be self-explanatory.

Messages indicating ‘‘compiler malfunction’’, or any error messages generated by the C compiler, indicate bugs in the SR compiler that should be reported to its implementors.

NAME

srl − SR linker

SYNOPSIS

srl [−eglvwA] [−I dir **] [−o** file **] [−r** file **] [−CLNOPQRVS** size **]** component ...

DESCRIPTION

Srl produces an executable file, **a.out** by default, by linking together the components of an SR program. The command line lists one or more resource and global names and may also list additional ‘*file*.o’, ‘*file*.a’, or ‘−l*xxx*’ arguments to link code from other languages. These additional arguments are passed to *ld*(1).

The last resource named becomes the main resource; it should have no parameters. Abstract resources (those having no bodies) should not be named, but all globals must be listed.

Srl accepts the following general options:

−I *dir* Use *dir*/Interfaces as the Interfaces directory instead of ./Interfaces.

−o *file* Use *file* for executable output instead of **a.out**.

−w Suppress warnings about out-of-date files.

−l List runtime limits, then exit immediately.

The following options set values controlling the subsequent execution of the SR program. For most maxima, the default value is 1000000.

−A Enable asynchronous output. When set, output calls do not block the entire program and are not atomic. Instead, they can affect the flow of control by performing context switches.

−C *n* Set the maximum number of active **co** statements.

−L *n* Set the maximum number of loop iterations between context switches. A value of zero is taken as infinite.

−N *n* Set the maximum number of **in** operation classes.

−O *n* Set the maximum number of active operations.

−P *n* Set the maximum number of processes.

−Q *n* Set the maximum number of pending remote requests.

−R *n* Set the maximum number of active resources.

−V *n* Set the maximum number of number of semaphores.

−S *size* Set the size of a process stack.

Additional options are useful mainly when debugging the SR system:

−e Use the experimental runtime system.

−r *file* Use the runtime system from *file.*

−g Link to allow debugging with *dbx*(1). This is useful mostly for debugging the runtime system and is not particularly helpful towards understanding errant SR programs.

−v Announce other programs invoked by *srl*.

ENVIRONMENT

SR_PATH

Srl searches for object files by looking first in the current directory, then in the Interfaces directory, and finally in the SR library. If SR_PATH is set, any directories named there are checked ahead of the standard search sequence. Directories in SR_PATH are separated by colons (**:**).

FILES

Interfaces/component.o	object file
srlib.a	runtime library
a.out	executable program
Interfaces/component.spec	specification file
Interfaces/_ofile.c	configuration information
Interfaces/_ofile.o	configuration object module

SEE ALSO

sr(1), ld(1), dbx(1)

DIAGNOSTICS

Srl does not actually do the linking itself; it is merely a front-end for the UNIX linker *ld*(1). Messages not beginning with ‘‘srl’’ can be attributed to *ld*.

Messages about undefined externals usually stem from the omission of a resource or global name from the *srl* call.

Srl verifies that object files of resource and globals are newer than the corresponding source files.

NAME

srm − SR makefile generator

SYNOPSIS

srm [options] file.sr ... [other files and libraries]

DESCRIPTION

Srm generates a makefile (for use with *make*(1)) that builds an SR program from the given *sr* source files (*.sr* files), other files (*.c*, *.o* files), libraries (-l*lib*), and archives (*.a* files).

Srm produces a makefile that uses *sr*(1) and *srl*(1) to compile, link, and execute the program described by the files and libraries listed on the command line. The generated makefile provides the following services:

make compile compiles without linking

make link links after a ‘‘make compile’’ (really the same thing as just ‘‘make’’)

make run executes the program (runtime arguments may be specified with the −**R** option)

make ls produces a list of the source files (e.g., ‘‘pr ‘make ls‘’’)

make clean removes all artifacts of compilation except for the executable

make cleanx removes all artifacts of compilation including the executable (additional files may be specified for removal with the −**Z** option)

make make re-makes the makefile as per the original instructions (useful if the resource import graph changes and the makefile needs to reflect the new dependencies)

The default makefile name is **Makefile**, and thus the command ‘‘make’’ executes it. The makefile name can be changed using the −**f** option. As a safety feature, *srm* refuses to overwrite an existing file that it did not create.

The default executable name is **a.out.** This can be changed using the −**o** option.

Srm attempts to determine the main resource (for linking purposes) by analyzing the source files and selecting the first resource it finds that is imported by the least number of resources. If some other resource is to be the main resource, it should be specified with the −**m** option.

Srm takes a number of option flags, which are specified in arguments beginning with a hyphen. The following options are recognized:

−**f** *mfile* Name the makefile *mfile* instead of **Makefile**. A file name of ‘‘−’’ directs the makefile to standard output.

−**I** *dir* Use *dir*/Interfaces as the Interfaces directory instead of ./Interfaces.

−**o** *ofile* Use *ofile* for executable output instead of **a.out**.

−**c** *compiler*
Use *compiler* instead of the default compiler.

−**m** *res* Use the specified resource as the main resource. The default is the first resource found that is imported by the fewest number of resources.

−**v** Display interesting information while building the makefile.

−**w** Output wide makefile lines instead of limiting to 78 columns.

−**C** *opts* Include *opts* in the list of options passed to the compiler *sr*(1).

−**L** *opts* Include *opts* in the list of options passed to the linker *srl*(1).

−**R** *args* Include *args* in the list of arguments passed to the executable by ‘‘make run’’.

−**Z** *files* Include *files* in the list of files removed by ‘‘make cleanx’’.

EXAMPLES

srm *.sr	no options, all .sr files in current directory
srm –m main –o x a.sr b.sr	the main resource and executable name are specified
srm –C "–q –O" a.sr b.sr	flags are passed to *sr*(1)
srm a.sr b.sr c.o –ll r.c –lm	objects, libraries and *C* sources listed are passed on to *srl*(1) to be linked in with the sr program.

ENVIRONMENT

SRMOPTS

The environment variable SRMOPTS may contain *srm* options to be processed ahead of explicit options.

SR_PATH

Srm searches for resources and object files by looking first in the current directory, then in the Interfaces directory, and finally in the SR library. (Items found somewhere other than the current directory are included in the makefile as *srl* arguments but not as dependencies.) If SR_PATH is set, any directories named there are checked ahead of the standard search sequence. Directories in SR_PATH are separated by colons (**:**).

FILES

Makefile	generated makefile
file.sr	SR source file
a.out	executable program
Interfaces/component.o	object file
Interfaces/resource.spec	export information

SEE ALSO

sr(1), srl(1), make(1)

DIAGNOSTICS

Srm uses the *sr* compiler to obtain dependency information. In addition to the syntax errors detected by the compiler, *srm* detects missing resource/global specifications and duplicate spec/body definitions. Warnings are issued if a resource or global has no body or a resource that is not the main resource is not imported by any other resource.

CAVEATS

Omitting *srm's* initial **s** can have disastrous results.

Srm doesn't escape shell meta-characters in the makefile.

If the main resource is imported by others, the **–m** option will probably be required to produce a correct makefile.

NAME

srprof − SR profiler

SYNOPSIS

srprof [−a] [tracefile **]**

DESCRIPTION

Srprof reads a trace file produced by an SR program and totals the counts of the events by line number. If the **−a** option is given, the report is produced in the form of an annotated program listing.

SEE ALSO

sr(1), srtrace(5)

CAVEATS

An annotated listing can be produced only when *srprof* is run in the directory containing the source files.

NAME

srgrind − format SR program for troff

SYNOPSIS

srgrind [−f] [−n] [−w] [−x] [−d *defs*] **[−h** *header*] **[−s** *size*] **[**file ...**]**

DESCRIPTION

Srgrind formats an SR program for typesetting by *troff*(1). *Srgrind* actually works by executing *vgrind*(1) with an appropriate set of files and parameters.

Input is read from the named files, or from standard input if none are given. The *troff* directives are written to standard output.

Srgrind accepts the following options:

−d *defs* Use an alternate *vgrindefs*(5) file instead of the default one inside *srgrind.*

−f Run in filter mode. In this mode, *srgrind* passes most input untouched, processing only those lines delimited by **.vS** and **.vE** macro calls.

−h *header* Place a header at the top of the page.

−n Don't embolden keywords.

−s *size* Set the pointsize for typesetting.

−w Consider tabs to be spaced every four columns instead of every eight (works only if supported by *vgrind).*

−x Pass **−x** to *vgrind* for index processing.

SEE ALSO

sr(1), troff(1), vgrind(1), vgrindefs(5), srtex(1)

CAVEATS

Srgrind is limited by the capabilities of *vgrind*. For example, comments of the form /∗...∗/ do not nest.

Vgrind macros can conflict with others when using filter mode.

Srgrind doesn't function on systems lacking *vgrind.*

NAME

srtex − format SR program for TeX

SYNOPSIS

srtex [−lp] [−a *n* **] [−t** *n* **] [−CIKS** *font* **] [** file ...]

DESCRIPTION

Srtex formats an SR program for *tex*(1), allowing the program to be typeset. It can produce output for either plain TeX (the default) or LaTeX. Input is read from the named files, or from standard input if none are given. Output is written to standard output.

Srtex tries hard to keep indentation and alignment the same as the original program. Horizontal positioning is recalculated (to preserve vertical columns) after reading a tab character or three consecutive spaces.

Srtex accepts the following options:

−l Produce LaTeX instead of TeX.

−p Produce a wrapper allowing output to be fed directly into TeX. By default, *srtex* produces output to be included in a larger document.

−t *n* Set tab stops every *n* columns. The default is every eight columns.

−a *n* Set the number of spaces that will force column alignment. The default is three. A large value inhibits alignment and usually looks ugly.

−C *font* Set the comment font; the default is "it".

−I *font* Set the identifier font; the default is "rm".

−K *font* Set the keyword font; the default is "bf".

−S *font* Set the string font; the default is "tt".

SEE ALSO

sr(1), tex(1), latex(1), srlatex(1), srgrind(1)

CAVEATS

Erroneous programs may exhibit strange spacing and/or pagination.

Extremely long program tokens will overflow lex buffers and cause core dumps.

It is possible to overflow TeX buffers.

NAME

srlatex − format SR program for LaTeX

SYNOPSIS

srlatex [−ep] [−l lang **] [−a** *n* **] [−t** *n* **] [** file ...]

DESCRIPTION

Srlatex formats an SR program for use with *latex(1).* To include an SR program formatted with *srlatex,* use document-style option *srlatex* and include the file with the output generated by *srlatex* with an *\input{ filename }* command. The document style option defines various parameters concerning the typesetting aspects. For example, fonts for keywords, identifiers, literal strings, and comments are defined there. Input is read from the named files, or from standard input if none are given. Output is written to standard output.

Srlatex tries hard to keep indentation and alignment the same as the original program. Horizontal positioning is recalculated (to preserve vertical columns) after reading a tab character or three consecutive spaces. The average character width used to calculate columns is fixed (as defined by *\srTeXcw* in the style file). If text exceeds this average length, a warning is printed and the text is typeset in its natural width.

The following options are accepted:

−e Enable escape sequence. Text between the delimiters #< and ># is passed to the typesetting program directly. The delimiters themselves are not printed. This allows to include arbitrary LaTeX commands in a program. Note that the SR compiler regards the whole line as a comment.

−l *lang* Select one of the following source languages:

- **sr** SR (Synchronizing Resources)
- **ftsr** Fault-Tolerant SR
- **pl** Programming Logic (from *Concurrent Programming*)
  PL adds the keywords **await**, **barrier**, **chan**, **cond**, **empty**, **minrank**, **monitor**, **region**, **signal**, **signal_all**, **wait**, and **when** to SR.

−p Produce a wrapper allowing output to be fed directly into LaTeX. By default, *srlatex* produces output to be included in a larger document.

−t *n* Set tab stops every *n* columns. The default is every eight columns.

−a *n* Set the number of spaces that force column alignment. The default is three. A large value inhibits alignment and usually looks ugly.

SEE ALSO

Gregory R. Andrews and Ronald A. Olsson, *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, 1993, ISBN 0-8053-0088-0.

Richard D. Schlichting and Vicraj T. Thomas, *FT-SR: A Programming Language for Constructing Fault-Tolerant Distributed Systems.* TR 92-31, Dept. of Computer Science, The University of Arizona, 1992.

Gregory R. Andrews, *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991, ISBN 0-8053-0086-4.

sr(1), latex(1), srtex(1), srgrind(1)

CAVEATS

Erroneous programs may exhibit strange spacing and/or pagination.

Extremely long program tokens overflow lex buffers and cause core dumps.

It is possible to overflow TeX buffers.

NAME

ccr2sr − CCR notation to SR code preprocessor

SYNOPSIS

ccr2sr [−e] file.ccr

DESCRIPTION

Ccr2sr converts a program written in CCR (Conditional Critical Region) notation into an equivalent one written in the SR language. The SR program is placed in file.sr, which may then be compiled and linked using *sr*(1). The **−e** (experimental) option is for testing *ccr2sr.*

The exact syntax of the CCR notation can be discerned by examining the CCR programs in the examples directory. Each CCR-notation construct begins with an underscore; CCR code is mixed with regular SR code. Syntactic peculiarities exist to ease implementation.

Ccr2sr translates a CCR resource into an SR global and translates a CCR region statement into code that uses Rem's algorithm. Typically, the rest of the user's code will be a single SR resource program. The resultant SR code should be run on only one virtual machine so that only one global is created. Variables declared within CCR resources can be used only within region statements.

One-dimensional and two-dimensional arrays of CCR resources are supported. The bounds and subscripts for arrays of CCR resources must be integers. In conventional CCR notation (e.g., see Andrews's book), each variable declared within a CCR resource is replicated within each element of the CCR resource array. In the *ccr2sr* CCR notation, such replication does *not* occur. Instead, to effect such replication, the programmer needs to augment the declaration of each variable declared within a CCR resource array with array bounds matching those for the CCR resource array; an element of these arrays should be accessed only within a region statement for the corresponding element of the CCR resource. Hence, former simple variables become arrays and former arrays become higher-dimensional arrays. A variable whose declaration is not augmented as described above will be shared between all elements of the CCR resource array, which is not conventional CCR semantics.

A number of syntactic limitations exist to ease the implementation. Do not use names of SR predefined functions or reserved words (e.g., free, exit, or skip) as variable names within CCRs. Do not use identifiers beginning with r_ within CCR code. Do not use any other SR synchronization or return/reply within a CCR program. Do not use the # form of comment; use only the /∗ ∗/ form, but do not nest comments. Do not use /% or %/. Spaces in source text, even within string literals, between the following pairs of characters will be deleted: / and %, % and /, / and /, [and], ∗ and ∗, ˜ and =, and : and =; in addition spaces preceding : and = will also be deleted. Within string literals, use an escape character (\) to retain the desired spacing. Do not use the [a,b] form of subscripting a two-dimensional array as part of a CCR construct (e.g., _region2); instead use the [a][b] form.

The predefined SR procedure nap, which puts the currently executing process to sleep for a while and performs a context switch to another process, is useful in CCR programs to alter the interleaving of process execution. For example, it can be used with the random number functions to obtain different interleavings to test whether a critical section algorithm works. See the programs in the examples directory for examples.

QUICK REFERENCE

_resource(*name*)
_resource_end(*name*)

_region(*name*,*when*) use **true** for *when* if empty
_region_end(*name*)

_resource1(*name*,*l1*,*u1*) one-dim CCR resource *name[l1:u1]*
_resource_end1(*name*)

_region1(*name*,*v1*,*when*) use **true** for *when* if empty; *v1* is subscript
_region_end1(*name*,*v1*)

_resource2(*name*,*l1*,*u1*,*l2*,*u2*) _resource_end2(*name*)	two-dim CCR resource *name[l1:u1,l2:u2]*
_region2(*name*,*v1*,*v2*,*when*) _region_end2(*name*,*v1*,*v2*)	use **true** for *when* if empty; *v1* and *v2* are subscripts

FILES

file.ccr	CCR source file
file.sr	generated SR source file

SEE ALSO

Gregory R. Andrews and Ronald A. Olsson, *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, 1993, ISBN 0-8053-0088-0.

Gregory R. Andrews, *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991, ISBN 0-8053-0086-4.

sr(1), cpp(1), csp2sr(1), m2sr(1)

DIAGNOSTICS

Some erroneous CCR code will cause errors from cpp. The line numbers that cpp complains about generally correspond to the ones in the .ccr file. Other than that, *ccr2sr* does nearly no error checking. To flag some errors, *ccr2sr* intentionally generates invalid SR code containing the word "ERROR" followed by an explanation. The SR compiler will detect that as an error later.

Other errors in CCR code are detected by the SR compiler. The line numbers for these errors will not correspond to those in the original source file, but they can be mapped back by looking in the generated code file. That file will be rather ugly, but the cause of the error can be found there. To give some help in tracing back errors to the original source file, the generated code file contains comments of the form /∗--X--∗/, where X is a line number in the original source file. These comments are generated for any construct that the preprocessor replaces by other text. If, for example, the SR compiler reports an error on line 38, then look at line 38 of the generated SR file. If on that line you find the comment /∗--12--∗/, examine line 12 of the original source file to see the error's cause. If you see no /∗--X--∗/ comment on the line itself, you should look for the first /∗--X--∗/ comment that appears above the line and use that number to point you back to the correct line in the original source file.

NAME

csp2sr − CSP notation to SR code preprocessor

SYNOPSIS

csp2sr [−ti] [−te] [−e] file.csp

DESCRIPTION

Csp2sr converts a program written in CSP (Communicating Sequential Processes) notation into an equivalent one written in the SR language. The SR program is placed in file.sr, which may then be compiled and linked using *sr*(1). The **−t** options specify which termination discipline is to be used. The **−e** (experimental) option is for testing *csp2sr.*

The exact syntax of the CSP notation can be discerned by examining the CSP programs in the examples directory. Each CSP-notation construct begins with an underscore; CSP code is mixed with regular SR code. Syntactic peculiarities exist to ease implementation.

Csp2sr translates a CSP program into an equivalent SR resource; the SR generated code uses a slight modification of the centralized clearing house technique described in Andrews's book. The generated code is not necessarily fair; i.e., ports are serviced nondeterministically.

Csp2sr supports implicit (i.e., automatic) or explicit termination disciplines. The choice is made by the corresponding command line option. The default is implicit termination. If a CSP input/output command that appears as a statement (e.g., _stmt_i but not _guard_i) fails, the entire program terminates. A process that attempts to send to or receive from itself is not detected as an error.

Following are a few restrictions in using *csp2sr* . CSP processes can have 0, 1, or 2 dimensions. The subscripts for arrays of processes must be integers. A process's spec must appear before its body. Code for all members of a process family must be the same. Nested processes are not allowed. SR operation names are used as constructors. There is no empty constructor. Operation names must be unique to the entire program, not just to a process. Parameters must be enclosed in '()' even if there are none. Operation declarations should not declare variable or result parameters, and should not include returns clause. Put no spaces around process or operation names in the CSP constructs.

The boolean part of a CSP guard can be only a single boolean expression; so, use '&' to separate parts of a guard (not ';' as in the CSP paper). Quantifier variables should not be declared. Quantifiers can have 1 or 2 variables. Variables cannot be declared as part of guards. The CSP **_if** and **_do** constructs do not allow an else part. The CSP **_do** construct does not allow exit or next statements.

A number of syntactic limitations exist to ease the implementation. Do not use names of SR predefined functions or reserved words (e.g., free, exit, or skip) as variable names within CSP code. Do not use identifiers beginning with csp_ within CSP code. Do not use any other SR synchronization or return/reply within a CSP program. Do not use the # form of comment; use only the /* */ form, but do not nest comments. Do not use /% or %/. Spaces in source text, even within string literals, between the following pairs of characters will be deleted: / and %, % and /, / and /, [and], * and *, ˜ and =, and : and =; in addition spaces preceding : and = will also be deleted. Within string literals, use an escape character (\) to retain the desired spacing. Do not use the [a,b] form of subscripting a two-dimensional array as part of a CSP construct (e.g., _guard); instead use the [a][b] form.

The predefined SR procedure nap, which puts the currently executing process to sleep for a while and performs a context switch to another process, is useful in CSP programs to alter the interleaving of process execution. For example, it can be used with the random number functions to obtain different interleavings to test whether a critical section algorithm works. See the programs in the examples directory for examples.

QUICK REFERENCE

_program(*name*)
_program_end

_specs	follow with process specs and port
_specs_end	

_dump_pidx	for implementation debugging
_process_spec(*name*)	*name* is a CSP process
_process_spec1(*name*, *l1*, *u1*)	one-dim CSP process array *name[l1:u1]*
_process_spec2(*name*, *l1*, *u1*, *l2*, *u2*)	two-dim CSP array *name[l1:u1,l2:u2]*
_port(*pname*,*oname*,*ospec*)	declares a CSP port

    pname, actually unused, is process that inputs from port
    oname is name of port
    ospec is parameters
    get one port for each element of an array

_process_body(*name*)	body of process *name*
_process_body1(*name*, *v1*)	one-dim body; *v1* is process id
_process_body2(*name*, *v1*, *v2*)	two-dim body; *v1*,*v2* is process id
_process_end	

_stmt_i(*pname*,*ouse*,*args*)
: input statement, i.e., *pname?ouse(args)*
: *pname* is source; *ouse* is port name; *args* are formals

_stmt_iq1(*v1*,*l1*,*u1*, *pname*,*ouse*,*args*)
: one-dim quantified input statement,
: i.e., *(v1 := l1 to u1) pname?ouse(args)*

_stmt_iq2(*v1*,*l1*,*u1*, *v2*,*l2*,*u2*, *pname*,*ouse*,*args*)
: two-dim quantified input statement
: i.e., *(v1 := l1 to u1, v2 := l2 to u2) pname?ouse(args)*

_stmt_o(*pname*,*ouse*,*args*)
: output statement, i.e., *pname!ouse(args)*
: *pname* is destination; *ouse* is port name; *args* are actuals

_stmt_oq1(*v1*,*l1*,*u1*, *pname*,*ouse*,*args*)
: one-dim quantified output statement
: i.e., *(v1 := l1 to u1) pname!ouse(args)*

_stmt_oq2(*v1*,*l1*,*u1*, *v2*,*l2*,*u2*, *pname*,*ouse*,*args*)
: two-dim quantified output statement
: i.e., *(v1 := l1 to u1, v2 := l2 to u2) pname!ouse(args)*

_if	CSP **if**, for using I/O in guards
don't separate guards with []	
_fi	
_do	CSP **do**, for using I/O in guards
don't separate guards with []	
_od	
_guard(*expr*)	plain boolean guard of **_if** or **_do**
_guard_q1(*v1*,*l1*,*u1*, *expr*)	one-dim quantified guard
i.e., *(v1 := l1 to u1) expr*	
_guard_q2(*v1*,*l1*,*u1*, *v2*,*l2*,*u2*, *expr*)	two-dim quantified guard
i.e., *(v1 := l1 to u1, v2 := l2 to u2) expr*	

_guard_i(*expr*,*pname*,*ouse*,*args*)
: input command as a guard of **_if** or **_do**; *expr* is boolean
: i.e., *expr; pname?ouse(args)*

_guard_iq1(*v1*,*l1*,*u1*, *expr*,*pname*,*ouse*,*args*)
: one-dim quantified input command as a guard of **_if** or **_do**
: i.e., *(v1 := l1 to u1) expr; pname?ouse(args)*

_guard_iq2(*v1*,*l1*,*u1*, *v2*,*l2*,*u2*, *expr*,*pname*,*ouse*,*args*)
two-dim quantified input command as a guard of **_if** or **_do**
i.e., *(v1 := l1 to u1, v2 := l2 to u2) expr; pname?ouse(args)*

_guard_o(*expr*,*pname*,*ouse*,*args*)
output command as a guard of **_if** or **_do**; *expr* is boolean
i.e., *expr; pname!ouse(args)*

_guard_oq1(*v1*,*l1*,*u1*, *expr*,*pname*,*ouse*,*args*)
one-dim quantified output command as a guard of **_if** or **_do**
i.e., *(v1 := l1 to u1) expr; pname!ouse(args)*

_guard_oq2(*v1*,*l1*,*u1*, *v2*,*l2*,*u2*, *expr*,*pname*,*ouse*,*args*)
two-dim quantified output command as a guard of **_if** or **_do**
i.e., *(v1 := l1 to u1, v2 := l2 to u2) expr; pname!ouse(args)*

FILES

file.csp	CSP notation source file
file.sr	generated SR source file

SEE ALSO

Gregory R. Andrews and Ronald A. Olsson, *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, 1993, ISBN 0-8053-0088-0.

Gregory R. Andrews, *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991, ISBN 0-8053-0086-4.

sr(1), cpp(1), ccr2sr(1), m2sr(1)

DIAGNOSTICS

Some erroneous CSP code will cause errors from cpp. The line numbers that cpp complains about generally correspond to the ones in the .csp file. Other than that, *csp2sr* does nearly no error checking. To flag some errors, *csp2sr* intentionally generates invalid SR code containing the word "ERROR" followed by an explanation. The SR compiler will detect that as an error later.

Other errors in CSP code are detected by the SR compiler. The line numbers for these errors will not correspond to those in the original source file, but they can be mapped back by looking in the generated code file. That file will be rather ugly, but the cause of the error can be found there. To give some help in tracing back errors to the original source file, the generated code file contains comments of the form /*--X--*/, where X is a line number in the original source file. These comments are generated for any construct that the preprocessor replaces by other text. If, for example, the SR compiler reports an error on line 38, then look at line 38 of the generated SR file. If on that line you find the comment /*--12--*/, examine line 12 of the original source file to see the error's cause. If you see no /*--X--*/ comment on the line itself, you should look for the first /*--X--*/ comment that appears above the line and use that number to point you back to the correct line in the original source file.

NAME

m2sr − monitor notation to SR code preprocessor

SYNOPSIS

m2sr [−sc] [−sw] [−su] [−sx] [−e] file.m

DESCRIPTION

M2sr converts a program written in monitor notation into an equivalent one written in the SR language. The SR program is placed in file.sr, which may then be compiled and linked using *sr*(1). The **−s** options specify which monitor signaling discipline is to be used. The **−e** (experimental) option is for testing *m2sr*.

The exact syntax of the monitor notation can be discerned by examining the monitor programs in the examples directory. Each monitor-notation construct begins with an underscore; monitor code is mixed with regular SR code. Syntactic peculiarities exist to ease implementation.

M2sr translates a monitor into an equivalent SR global; the SR generated code uses a slight modification of the technique described in Joe Herman's thesis. Typically, the rest of the user's code will be a single SR resource program. Monitor operations must be declared in the monitor's specification part. User code needs to import the monitor global; it can invoke monitor operations via call statements (in which the operation name is qualified by the name of the monitor, as per usual SR rules).

M2sr supports SC (signal and continue), SX (signal and exit), SW (signal and wait), and SU (signal and urgent wait) signaling disciplines. The choice is made by the corresponding command line option. The default is SC.

The operations on condition variables are the standard ones: wait(cv), signal(cv), pri_wait(cv,rank), empty(cv), minrank(cv), and signal_all(cv). (signal_all only makes sense in the SC signaling discipline; it is not allowed in the others). In addition, print(cv), intended for primitive debugging, outputs the number of processes waiting on the condition variable and their ranks. Arrays of one or two dimensions of condition variables can be declared, although the syntax is baroque; e.g., condvar1(scan,0:1) declares scan to be a one dimensional array with indices 0 and 1, and condvar2(foo,3,5:9) declares foo to be a two dimensional array with indices 1 through 3 in the first dimension and 5 through 9 in the second dimension.

A number of syntactic limitations exist to ease the implementation. Do not use names of SR predefined functions or reserved words (e.g., free, exit, or skip) as variable names within monitors. Do not use identifiers beginning with m_ within monitor code. Do not use any other SR synchronization or return/reply within a monitor program. Do not use the # form of comment; use only the /∗ ∗/ form, but do not nest comments. Do not use /% or %/. Spaces in source text, even within string literals, between the following pairs of characters will be deleted: / and %, % and /, / and /, [and], ∗ and ∗, ˜ and =, and : and =; in addition spaces preceding : and = will also be deleted. Within string literals, use an escape character (\) to retain the desired spacing. Do not use the [a,b] form of subscripting a two-dimensional array within a monitor construct (e.g., _wait); instead use the [a][b] form.

The predefined SR procedure nap, which puts the currently executing process to sleep for a while and performs a context switch to another process, is useful in monitor programs to alter the interleaving of process execution. For example, it can be used with the random number functions to obtain different interleavings to test whether a critical section algorithm works. See the programs in the examples directory for examples.

QUICK REFERENCE

_monitor(*name*)	(after which, declare each monitor proc as an op)
_body(*name*)	
_monitor_end	
_condvar(*x*)	declare condition variable *x*
_condvar1(*x*,*s*)	declare one-dim array condition variable *x[s]*
_condvar2(*x*,*s*,*t*)	declare two-dim array condition variable *x[s,t]*
_proc(*x*)	monitor procedure with name and parameters *x*
_proc_end	

_wait(*cv*)	
_empty(*cv*)	
_pri_wait(*cv*,*r*)	prioritized wait by rank *r* on condition variable *cv*
_minrank(*cv*)	
_print(*cv*)	(for debugging; not a regular monitor primitive)
_signal(*cv*)	
_signal_all(*cv*)	(only for SC signaling discipline)

FILES

file.m	monitor notation source file
file.sr	generated SR source file

SEE ALSO

Gregory R. Andrews and Ronald A. Olsson, *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings, 1993, ISBN 0-8053-0088-0.

Gregory R. Andrews, *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991, ISBN 0-8053-0086-4.

sr(1), cpp(1), ccr2sr(1), csp2sr(1)

DIAGNOSTICS

Some erroneous monitor code will cause errors from cpp. The line numbers that cpp complains about generally correspond to the ones in the .m file. Other than that, *m2sr* does nearly no error checking. To flag some errors, *m2sr* intentionally generates invalid SR code containing the word "ERROR" followed by an explanation. The SR compiler will detect that as an error later.

Other errors in monitor code are detected by the SR compiler. The line numbers for these errors will not correspond to those in the original source file, but they can be mapped back by looking in the generated code file. That file will be rather ugly, but the cause of the error can be found there. To give some help in tracing back errors to the original source file, the generated code file contains comments of the form /*--X--*/, where X is a line number in the original source file. These comments are generated for any construct that the preprocessor replaces by other text. If, for example, the SR compiler reports an error on line 38, then look at line 38 of the generated SR file. If on that line you find the comment /*--12--*/, examine line 12 of the original source file to see the error's cause. If you see no /*--X--*/ comment on the line itself, you should look for the first /*--X--*/ comment that appears above the line and use that number to point you back to the correct line in the original source file.

In some cases, the generated code for monitor code using the SX signaling discipline causes complaints from the SR compiler, or later from the C compiler, about unreachable code. These warnings can be ignored.

NAME

srv − verify correct functioning of SR system

SYNOPSIS

srv [−option ... **] [** dir ... **]**

DESCRIPTION

Srv executes verification scripts in subdirectories of the SR verification suite. It is part of the SR installation process and is not used by SR programmers.

If one or more *dir* arguments are given, testing is restricted to those directories and descendents; otherwise the entire suite is searched. Within each subdirectory, a **Script** file directs the verification process. Each line contains an expected status code followed by a command. If the command returns an unexpected status, the test fails and (except for a **run** command) the script is abandoned. Empty lines, and lines beginning with **#**, are ignored.

For **sr**, **srl**, **srm**, **srprof**, **srgrind**, **srtex**, **srlatex**, **ccr2sr**, **m2sr**, and **csp2sr** commands, production or experimental versions are selected according to *srv* options, and output is redirected to reserved file names. A **run** command executes **a.out**, redirecting standard input if a file is named; output is directed to a related file and the results are compared with what was expected. Differences report a failure and disable **rm** commands but do not abandon the script. Other commands are simply executed by the shell with no special handling.

Srv normally finds binaries in the source hierarchy. The **−p** option selects instead the production (installed) version of the system. The options **−c**, **−l**, **−r**, and **−t** individually select the production versions of the compiler, linker, runtime system, and other tools respectively.

The **−v** option echoes each Script line as it is read.

FILES

Script	verification script
Compiler.out	compiler output
Linker.out	linker output
Maker.out	makefile maker output
Profiler.out	profiler output
Grinder.out	troff grinder output
Texer.out	TeXer output
Latexer.out	LaTeXer output
CCR.out	ccr2sr output
M.out	m2sr output
CSP.out	csp2sr output
xxxxx.std	expected output from input file ‘xxxxx’
xxxxx.out	actual output from input file ‘xxxxx’

SEE ALSO

srvi(1), sr(1), srl(1), srm(1), srprof(1), srgrind(1), srtex(1), srlatex(1), ccr2sr(1), m2sr(1), csp2sr(1).

DIAGNOSTICS

Srv lists the SR system files to be used, and aborts if any are inaccessible. Each directory is listed as testing begins. Any additional messages indicate a test failure.

CAVEATS

Srv does not detect infinite loops.

A full run starts slowly due to pipeline delays.

Quoted arguments don’t work with the specially recognized commands.

NAME

srvi − install an SR program in the verification suite

SYNOPSIS

srvi [−option ...] dir

DESCRIPTION

Srvi installs an SR program in the SR verification suite. It is part of the SR installation process and is not used by SR programmers.

A new directory *dir* is created in the vsuite hierarchy to hold the program and associated files. *dir* may be given as a path relative to vsuite if the intermediate directories exist.

Srvi prompts for source files and input files. Enter each of these as a space-separated list. The list of input files may be empty.

After the specifications have been entered, the files are copied into the vsuite hierarchy and the program is compiled and linked. If no errors are found, it is executed using each specified input file in turn as standard input. If no input files are given, it is executed once with input from /dev/null. Standard output and error files are combined and saved from each run. A **Script** file is created for *srv*(1).

Srvi normally finds binaries in the source hierarchy. The **−c** and **−r** options select the production (installed) versions of the compiler and runtime system respectively; the **−p** option selects both of these.

FILES

All files are created in vsuite/*dir*.

Script	verification script
Compiler.std	compilation messages, if any
xxxxx.std	output from input file ‘xxxxx’

SEE ALSO

srv(1), sr(1), srl(1)

DIAGNOSTICS

Srvi lists the SR system files to be used, and aborts if any are inaccessible. It checks the accessibility of all source and data files before copying anything or creating the **Script** file.

Srvi refuses to overwrite a directory with an existing **Script** file.

CAVEATS

Srvi does not support the full generality of *srv*. Complex verification scripts must be created manually.

NAME

sranimator − XTANGO based animation package for SR

SYNOPSIS

(in SR program:)

import SRanimator

(to build:)

sr program.sr **sranimator.o** *XTANGO*/**xtango.o −lXaw −lXmu −lXext −lXt −lX11**

(with srm:)

srm program.sr **sranimator.o** *XTANGO*/**xtango.o −lXaw −lXmu −lXext −lXt −lX11**

(to run:)

xrdb −merge *XTANGO*/**xtango.res**

a.out

DESCRIPTION

SRanimator provides a front-end to XTANGO, a library of algorithm animation routines. The SR application imports a global **SRanimator** and links with SR, XTANGO, and X library files. The library lists shown above assume that XTANGO was built using the Athena widget set.

RUNNING AN ANIMATION

Before running an XTANGO application you must run **xrdb −merge** *XTANGO*/**xtango.res**, where **xtango.res** is the application default file from the XTANGO distribution. Alternative methods of initialization are discussed in the XTANGO documentation.

To run an animation, compile and run the SR application containing the procedure calls described below. When the animation window appears, press the **run animation** button to begin the animation.

Animation parameters can be altered using other pushbuttons either before or after the animation has been started. The arrows and zoom pushbuttons on the left control the display, while the buttons at the bottom and the scrollbar at the right affect the animation behavior. An animation may be aborted at any time by selecting the **quit** pushbutton in the lower right-hand corner of the animation window.

Panning and Zooming

The pushbuttons along the left-hand edge of the animation screen affect what portions of an animation are displayed while an animation is running. To pan the animation window in a specific direction, select the appropriate arrow pushbutton. To zoom in on a section of the animation select the **in** pushbutton; select the **out** pushbutton to zoom back out.

Pausing a Running Animation

After initially selecting the **run animation** pushbutton to start the animation running, the **run animation** pushbutton changes its name to **pause**. An animation may be paused at any time by selecting the **pause** pushbutton which then pauses the animation and changes its name to **unpause**. To continue the animation, select the **unpause** pushbutton which then changes its name back to **pause**.

Changing the Animation Refresh Mode

When using Motif and HP widgets, XTANGO supports three different times to redraw all images displayed in an animation: between each frame, between each scene, and none. The default refresh mode is between each *frame*. (With Athena widgets, the mode setting has no effect.)

There is a direct trade-off between the frequency of redrawing and the speed of the animation. Refreshing **by frame** produces a smooth animation but runs the slowest. On the other hand, **no refresh** produces a quick animation possibly with many holes in the images (created when one image passes over another). Try the **no refresh** mode when initially building an animation. Then switch to **by scene** to iron out the minor bugs. Finally use the **by frame** mode when an animation is complete.

To change the refresh mode, select the **mode** pushbutton either before running an animation or during the animation. A menu will appear containing the three choices and the current choice will be indicated with a checkmark. After selecting the appropriate choice, the new refresh mode will take effect immediately.

Select the **refresh** pushbutton located next to the **quit** pushbutton to force a refresh at any time.

Slowing Down a Running Animation

Use the vertical scrollbar located along the right-hand edge of the animation window to control the amount of time between animation frames. The animation runs fastest when the scrollbar's ''thumb'' (the black, movable rectangle) is positioned at the top of the scrollbar; the animation becomes progressively slower as the thumb is moved towards the bottom of the scrollbar.

To change the position of the thumb, either select the thumb and drag it to a new location, select a location within the scrollbar (and the thumb will jump there), or select one of the arrows located at the ends of the scrollbar (and the thumb will step in the desired direction).

INTERFACE PROCEDURES

Animations operate on an infinite plane having a real-valued coordinate system. A rectangular portion of the plane, initially the area having x and y values between 0.0 and 1.0, is displayed on the screen. Graphical objects can be created and placed within the coordinate system, and then moved or altered to depict the operations and actions of a computer algorithm. The SR programmer specifies a unique integer *id* when creating an object, then uses this *id* to designate the object in subsequent operations.

The individual animation commands are described below. Most commands and parameters should be self-explanatory. Arguments named *steps*, *centered*, and *steptime* are of integer type. Arguments named *xpos*, *ypos*, *xsize*, *ysize*, *radius*, *lx*, *by*, *rx*, *ty*, *ry* are real or floating point numbers. The argument *fillval* should be one of the following strings: **outline**, **light**, **half**, **heavy**, or **solid**. The argument *widthval* should be one of the following strings: **thin**, **medthick**, or **thick**. The argument *styleval* should be one of the following strings: **dotted**, **dashed**, or **solid**. The argument *arrows* should be one of the following strings: **none**, **forward**, **backward**, or **bidirectional**. The parameter *colorval* can be any color specification acceptable to X. Note that the *color* command only supports a subset of all these possible colors: **white**, **black**, **red**, **orange**, **yellow**, **green**, **blue**, and **maroon**. The parameter *fontname* can be any font specification acceptable to X.

General Procedures

A_bg (colorval : string[∗])

Change the background to the given color. The default background is white.

A_coords (lx, by, rx, ty : real)

Change the displayed coordinates to the given values (left-bottom is (*lx*,*by*) and right-top is (*rx*,*ty*)). You can use repeated applications of this command to pan or zoom the animation view.

A_delay (steps : int)

Generate the given number of animation frames with no changes in them.

A_zoom (id : int; rx, ry: real)

Zoom in to (positive values of *rx* and *ry*) or out from (negative values) the object given by *id*. Values close to 0.0 zoom a little, absolute values close to 1.0 zoom a lot.

A_end ()

Terminate the animation.

Drawing Procedures

A_line (id : int; xpos, ypos, xsize, ysize : real; colorval, widthval, styleval, arrows : string[∗])

Create a line with one endpoint at the given position and of the given size. The line can be dotted, dashed, or solid and can optionally have arrows on either or both ends. Note that lines are moved (move, jump, and exchange commands) relative to their centers.

A_rectangle (id : int; xpos, ypos, xsize, ysize : real; colorval, fillval : string[∗])

Create a rectangle with lower left corner at the given position and of the given size (size must be positive).

A_circle (id : int; xpos, ypos, radius : real; colorval, fillval : string[∗])

Create a circle centered at the given position.

A_triangle (id : int; v1x, v1y, v2x, v2y, v3x, v3y : real; colorval, fillval : string[∗])

Create a triangle whose three vertices are located at the given three coordinates. Note that triangles are moved (move, jump, and exchange commands) relative to the centers of their bounding boxes.

A_text (id : int; xpos, ypos : real; centered : int; colorval, str : string[∗])

Create text *str* with lower left corner at the given position if *centered* is 0. If *centered* is 1, the position arguments denote the place where the center of the text is put. The text string is allowed to have blank spaces included in it but you should make sure it includes at least one non-blank character.

A_bigtext (id : int; xpos, ypos : real; centered : int; colorval, str : string[∗])

This works just like the text command except that this text is in a much larger font.

A_fonttext (id : int; xpos, ypos : real; centered : int; colorval, fontname, str : string[∗])

This works just like the text command except that this text is in the specified font.

Image Manipulation Procedures

A_move (id : int; xpos, ypos : real)

Smoothly move, via a sequence of intermediate steps, the object with the given id to the specified position.

A_moverelative (id : int; xdelta, ydelta : real)

Smoothly move, via a sequence of intermediate steps, the object with the given id by the given relative distance.

A_moveto (id1, id2 : int)

Smoothly move, via a sequence of intermediate steps, the object with the first id to the current position of the object with the second id.

A_jump (id : int; xpos, ypos : real)

Move the object with the given id to the designated position in a one frame jump.

A_jumprelative (id : int; xdelta, ydelta : real)

Move the object with the given id by the provided relative distance in one jump.

A_jumpto (id1, id2 : int)

Move the object with the given id to the current position of the object with the second id in a one frame jump.

A_stepjump (id : int; xpos, ypos : real; steps, steptime : int)

Move the object with the given id to the designated position in a multiple frame jump. The steps of the jump are done at the specified millisecond intervals.

A_stepjumpto (id1, id2 : int; steps, steptime : int)

Move the object with the given id to the current position of the object with the second id in a multiple frame jump. The steps of the jump are done at the specified millisecond intervals.

A_color (id : int; colorval : string[∗])

Change the color of the object with the given id to the specified color value. Only the colors white, black, red, green, blue, orange, maroon, and yellow are valid for this command.

A_delete (id : int)

Permanently remove the object with the given id from the display, and remove any association of this id number with the object.

A_fill (id : int; fillval : string[*])

Change the object with the given id to the designated fill value. This has no effect on lines and text.

A_vis (id : int)

Toggle the visibility of the object with the given id.

A_lower (id : int)

Push the object with the given id backward to the viewing plane farthest from the viewer.

A_raise (id : int)

Pop the object with the given id forward to the viewing plane closest to the viewer.

A_exchangepos (id1, id2 : int)

Make the two objects specified by the given ids smoothly exchange positions.

A_switchpos (id1, id2 : int)

Make the two objects specified by the given ids exchange positions in one instantaneous jump.

A_swapid (id1, id2 : int)

Exchange the ids used to designate the two given objects.

A_resize (id : int; rx, ry: real)

The circle, line, rectangle, or triangle is resized as follows. The radius of a circle has *rx* added to it, the endpoint of a line has (*rx*,*ry*) added to it, the lower-right corner of a rectangle is dragged by amount (*rx*,*ry*), and the first vertex of a triangle is dragged by amount (*rx*,*ry*).

FILES

SRanimator.sr	SR animator global resource
animator.o	compiled C language animator commands
xtango.o	compiled C language XTANGO library
xtango.res	XTANGO widget resources for X11 resources database

SEE ALSO

sr(1), srl(1)

Stephen J. Hartley, *Animating Operating Systems Algorithms with XTANGO.* ACM SIGCSE Bulletin 26, 1 (March, 1994).

Stephen J. Hartley, *Integrating XTANGO's Animator into the SR Concurrent Programming Language.* Submitted for publication, 1994; included in the SR distribution.

John T. Stasko, *Tango: A Framework and System for Algorithm Animation.* IEEE Computer 23, 9 (September, 1990), 27-39.

Doug Hayes, *The XTANGO Environment and Differences from TANGO.* John T. Stasko and Doug Hayes, *XTANGO Algorithm Animation Designer's Package.* These two papers are provided as part of the XTANGO package.

CAVEATS

Bracketing blocks of animation code with *setpriority(1)* and *setpriority(0)* may improve the animation.

The XTANGO package must be obtained and built separately in order to use the SR animator. XTANGO is available by anonymous FTP from **par.cc.gatech.edu**. The SR library must be built (or rebuilt) after installing XTANGO.

AUTHOR

Stephen J. Hartley

ACKNOWLEDGMENTS

SRanimator was inspired by SRWin, written by Qiang A. Zhao, and by the **animator** interpreter program, included with XTANGO, written by John T. Stasko and Doug Hayes.

NAME

srgetopt − parse command arguments

SYNOPSIS

import SRgetopt
do c := getopt(*optstring***) != EOF –>** *process option c* **od**

DESCRIPTION

SRgetopt provides a means for an SR program to parse command line arguments in accordance with the standard Unix conventions; it is analogous to, and based on, *getopt*(3) for C programs. *SRgetopt* is a global containing one procedure, *getopt,* and variables that control its behavior or return additional information.

Getopt interprets command arguments in accordance with the standard Unix conventions: option arguments are introduced by "–" followed by a key character; an argument value follows certain keys. Multiple keys can be combined, as in "−ab", if they do not require arguments. A non-option argument terminates the processing of options, as does the special argument "−−".

Option interpretation is controlled by the parameter *optstring,* which specifies the characters that designate legal options and indicates which ones require associated values. The call **getopt("ab")** specifies that the command line should contain only the options "–a" and "–b". If a letter in *optstring* is followed by a colon, the option is expected to have an argument. The argument may or may not be separated by whitespace from the option letter. For example, **getopt("w:")** accepts either "–w 80" or "–w80".

Each call to *getopt* returns the key of the next command line argument; this key must match a letter in *optstring.* If the option accepts an argument, the string variable *optarg* is set to the argument value. Predefined conversion functions such as *int*, *char*, etc. can then be applied to *optarg.* The constant *optMAXLEN* defines the length of the longest string that *getopt* can handle; extra characters are truncated silently.

Getopt places in the variable *optind* the index of the next command line argument to be processed; *optind* is automatically initialized to 1 before the first call to *getopt.*

When all options have been processed, and only non-option arguments remain, *getopt* returns *optEOF.* The remaining arguments can be retrieved using the predefined function *getarg*, beginning with argument number *optind.*

DIAGNOSTICS

Getopt prints an error message on **stderr** and returns a question mark (′?′) when it encounters a command line argument not matched by *optstring.* Setting the variable *opterr* to **false** disables this error message.

NOTES

The following notes describe *SRgetopt*'s behavior in a few interesting or special cases; this behavior is consistent with *getopt*(3).

A ′–′ by itself is treated as a non-option argument. By convention, most programs interpret this as specifying the use of **stdin** or **stdout**, depending on context, in place of a named file.

If *optstring* is "a:" and the command line arguments are "–a –x", then "–x" is treated as the argument associated with the "–a".

Duplicate command line options are allowed; it is up to user to deal with them appropriately.

An option ‘‘letter’’ can be a letter, number, or almost any special character. Like *getopt*(3), *SRgetopt* disallows a colon as an option letter.

EXAMPLE

The following code fragment shows how to use *SRgetopt* to process a command that takes the options ′a′, ′f′, and ′w′ where ′f′ is followed by a file name and ′w′ is followed by an integer.

```
resource main()

    import SRgetopt
    var ch: char

    # command line arguments
    var aflg := 0
    var filename: string[optMAXLEN] := "out"
    var width := 80

    do (ch := getopt("abf:w:")) != optEOF –>
        if ch = ′a′ –>
            aflg++
        [] ch = ′f′ –>
            filename := optarg
        [] ch = ′w′ –>
            width := int(optarg)
        [] else –>
            stop(1)
        fi
    od

    write("−a", aflg)
    write("−f", filename)
    write("−w", width)

    fa k := optind to numargs() –>
        var xx: string[40]
        getarg(k,xx)
        write("normal argument", k, "is", xx)
    af

end
```

SEE ALSO

getopt(3)

CAVEATS

Changing the value of the variable *optind* may lead to unexpected behavior.

Getopt, like the predefined functions *numargs* and *getarg*, is valid only on the main virtual machine.

NAME

srwin − X-Windows graphics interface for SR

SYNOPSIS

(in SR program:)	**import SRWin**
(to build:)	**sr** program.sr **srwin.o –lX11**
(with srm:)	**srm** program.sr **srwin.o –lX11**

DESCRIPTION

SRWin provides a set of window interface operations to the SR programmer. It is based on the Xlib interface to the X window system. The application imports a global **SRWin** and links with a file of associated C functions and the X library. The library is often found simply by linking **–lX11**, but some systems may require a different specification.

The package supports five basic types of objects: windows, events, fonts, cursors, and images. Windows are primary objects for input and output. Cursors and fonts are resources of the underlying X Window System, which define the visible shape of the pointing device and the appearance of textual output respectively. Window events are queued in the order of their occurrence and handled later either by the SRWin package or by the application program.

Windows

Windows can be displayed on a X server or they can be off-screen pixmaps. To the programmer they are obscured data structures. They reside in the X server's memory, which means drawings on windows must be packed as requests and sent to the X server (probably through the network). A window cannot be shared among processes on different virtual machines.

A backup copy of each window's contents is maintained in the package for automatic refreshing when needed. A window can be treated as an abstract drawing surface without regard to overlap or visibility considerations.

Usually a graphics primitive does not contain all the information needed to draw a particular thing. The X server maintains resources called *graphics contexts (GCs)* that specify many attributes that apply to each graphic request. GCs reduce the traffic between the X server and the application programs (*clients* of the X server) in two ways. First, the GC information is maintained by the X server and only needs to be sent once, and later drawing requests use the id of the GC to draw. In the case of changes, only the selected few fields need to be sent. Second, multiple GCs can be created so clients can switch among different graphic bindings with ease.

SRWin treats GCs as parts of a window. The application program creates a new GC binding by calling **WinNewContext**() to get a new window structure with the new GC, while it still refers to the old window.

A window can have many *subwindows* as its children. All children are displayed on top of their parents when mapped (which means ‘‘ready for display’’). When the extent of a window intersects with that of a sibling window, these two windows can be displayed in any order. SRWin provides a few primitives for managing subwindows.

Events

All inputs from a window are modeled as events. Events that require a part of the screen to be redrawn are handled automatically by the SRWin package. Other events can be processed by the application program. Unlike most other X Window packages, with which application programs play a passive role responding to events, SRWin lets an application program manage the control flow directly.

To receive events on a window, the application program selects events of interest by specifying an event mask. When an event occurs, it is checked with the window's mask; if the particular event not selected, it is propagated to the parent window, and so on, until it reaches a window for which the event is selected or ignored (in which case the event is discarded).

Images

Images are rectangular areas of pixels; unlike windows, they reside on the client instead of the server, and they are always invisible. Images support only a limited set of operations, but image modification operations can be accomplished very quickly because they require no communication with the server. The typical use is to constuct an image locally, possibly a pixel at a time, and then copy it to the screen as a unit.

Fonts and Cursors

A cursor is a transient shape that moves on the screen as the mouse moves on its pad to indicate where the mouse is pointing. Different windows may have different cursor shapes.

A font is a set of bitmaps representing characters, cursor shapes, or other small patterns. In SRWin, fonts and cursors come from a predefined set supported by the underlying X Window System.

OUTPUT MODEL

All window output operations are defined in terms of an integer coordinate system with its origin at the top-left corner of the window.

Lines, ellipses, texts, and individual pixels can be drawn. Multiple characteristics of drawing and filling can be controlled. These include line width, line style, cap style, join style, dashes and dash offset, fill style and fill rule.

To further reduce the communication traffic between the X server and the application program, drawing can be temporarily disabled (on a per top-level window basis). Then the client must explicitly tell the X server to update the window contents. Alternatively the application can choose to construct an image and send the image as a whole to the X server for display.

INPUT MODEL

All input accepted by the package is represented by a uniform event record data structure, which includes information about:

- type of the event
- window in which the event occurred
- coordinates of the pointer within the window when the event occurred
- the status of the mouse buttons (up or down)
- the status of the SHIFT, CONTROL, and META keys
- specific values associated with the particular event type (e.g. button number when it is a button press or release event; ASCII code of keypress; etc.)

Each event from a window is dispatched to the event-message channel of that window. This channel is provided by the user program when it calls SRWin to create a window. If the user program doesn't provide an event channel, the window can be used only for output.

The user program can determine whether there is event pending by checking the number of messages in the event channel, or it can wait on the event channel until an event happens. The application can specify the event types of interest and have all other events discarded automatically. A process is generated for each window to poll for events.

EVENT CHANNEL

The *SRWin* global exports

optype *winEventChannel*(*winEvent*) **{send}**

The application program can use this optype to declare a message channel.

When a selected event occurs, SRWin sends a *winEvent* record to the registered input channel for the window. The *winEvent* record contains the following fields:

event_type	Type of the event.
window	The window in which the event occurred.
x, y	Coordinates of the pointer within the window when the event occurred.

bk_status	Inclusive OR of flags indicating the currently pressed buttons and keys:		
	BK_None	*BK_Button1*	*BK_Mod1*
	BK_CNTRL	*BK_Button2*	*BK_Mod2*
	BK_LOCK	*BK_Button3*	*BK_Mod3*
	BK_SHIFT	*BK_Button4*	*BK_Mod4*
		BK_Button5	*BK_Mod5*

data For a key event, *data* specifies which key or button was pressed or released, and it can be converted to the corresponding character. For a mouse button event, *data* is one of the button masks listed above. For a enter/leave window event, *data* can be converted to a boolean value indicating whether the window has the focus or not. For all other events, this field is undefined.

keysym Numerical value of standard X KeySym as defined in C header file *<X11/keysymdef.h>* and the KeySym database */usr/lib/X11/XKeysymDB*. It is useful for detecting keys that do not have corresponding ASCII character representations, such as function keys and arrow keys.

OTHER DATA STRUCTURES

winWindow Pointer to a record structure that holds all information for a window.

winInitialState Integer value specifying the initial state of an object when creating it.

winError Integer value that is zero in the case of an error or nonzero if successful.

winStdCursor An enumeration of the set of defined cursor shapes.

winCursor Pointer representing the handle of a cursor.

winColor String containing a color name or a numerical color specification.

winPixel Pointer representing the handle of a colormap entry.

winFont Pointer to a structure that holds information for a loaded font.

winImage Pointer to an image structure.

winPoint Record of (*x*, *y*) coordinates.

winRectangle Record of (*x*, *y*, *w*, *h*) for the coordinates of the top-left corner of the rectangle and its width and height.

winLineStyle Enumeration of valid line styles: *LineSolid, LineDoubleDash, LineOnOffDash*

winCapStyle Enumeration of valid cap styles: *CapNotLast, CapButt, CapRound, CapProjecting*

winJoinStyle Enumeration of valid join styles: *JoinMiter, JoinRound, JoinBevel*

winFillStyle Enumeration of valid fill styles: *FillSolid, FillTiled, FillOpaqueStippled, FillStippled*

winFillRule Enumeration of valid fill rules: *FillEvenOddRule, FillWindingRule*

winArcMode Enumeration of valid arc modes: *ArcChord, ArcPieSlice*

winDrawOp Enumeration of valid drawing operations: these control how the source pixel values generated by a graphics request are combined with the old destination pixel values already on the screen to produce the final destination pixel values. The operations are:

Op_Clear	*Op_And*	*Op_AndReverse*	*Op_Copy*
Op_AndInverted	*Op_Noop*	*Op_Xor*	*Op_Or*
Op_Nor	*Op_Equiv*	*Op_Invert*	*Op_OrReverse*
Op_CopyInverted	*Op_OrInverted*	*Op_Nand*	*Op_Set*

NAMING AND ARGUMENT CONVENTIONS

SRWin follows a set of conventions for the naming and syntax of the functions:

- The names of all SRWin functions begin with *Win* followed by compound words which are constructed by capitalizing the first letter of each word.

- Names of user-visible data structures and types begin with *win*. Names of all members of data structures use lower case.
- The window argument, where used, is always first in the argument list. The image argument, where used, is always right after the window argument when there is one, or the first when there is no window argument.
- Source arguments always precede destination arguments in an argument list.
- An *x* argument always precedes a *y* argument in an argument list.
- A *width* argument always precedes a *height* argument in an argument list.
- If *x*, *y*, *width*, and *height* arguments are used together, the *x* and *y* arguments always precede the *width* and *height* arguments.
- If a procedure returns an integer, a value of zero serves as an error indicator. If a procedure returns a pointer, **null** indicates an error. Not all errors are reported in this manner; some (especially those that cannot be detected immediately) abort the program.

FUNCTIONS

General Functions

WinOpen (display: string[*]; title: string[*]; evchannel: cap winEventChannel; state: winInitialState; w, h: int) returns win: winWindow

WinCreateSubwindow (oldwin: winWindow; evchannel: cap winEventChannel; state: winInitialState; x, y, w, h: int) returns newwin: winWindow

WinOpen() opens and initializes a top-level window of width *w* pixels, height *h* pixels, and with the same depth of the root window, with white foreground and black background. If **WinOpen**() can't open such a window, a **null** pointer is returned. **WinCreateSubwindow**() creates a subwindow as *oldwin*'s child. The subwindow begins at *(x, y)* relative to its parent's top-left corner.

The initialization includes opening a connection to the X server, creating a window, creating a backing store, allocating a graphics context and a colormap, loading default font, setting default window attributes, etc. If the *display* argument is a null string, SRWin then tries to open that window on the screen specified by the environment variable **DISPLAY**.

When *state* equals to **UseDefault**, the created window is displayed on screen (at a position determined by the window manager for the top-level window case), and output to the window is enabled. If *state* equals to **OffScreen**, the window is off screen and can be made visible by calling **WinMapWindow**(), while direct output is initially disabled.

The *evchannel* argument is used to register a message channel to receive incoming window events. It can be **null** if no event reporting is wanted. If the window is on screen at the beginning and *evchannel* is not **null**, then all events are selected on this window.

For a subwindow, the graphics context information is inherited from its parent, but in a different GC.

WinDestroyWindow (win: winWindow)

Destroys a window and all its subwindows, freeing contexts. This operation has no effect on a top-level window.

WinClose (win: winWindow)

Destroys a top-level window and all its subwindows, frees the associated X resources, and closes its connection to the X server.

WinNewContext (oldwin: winWindow) returns newwin: winWindow

Creates a new context window from an existing window. The context window appears as a ''window'' and points to the original window except it cannot generate any window events and it has a different graphics context.

WinCopyContext (srcwin, destwin: winWindow)

Copies all information associated with *srcwin*'s graphics context to that of *destwin*'s.

WinSetBorder (win: winWindow; width: int; color: winColor)

Sets the window border *width* and paints it using *color*. The border is not included when creating a window.

WinSetLabels (win: winWindow; winlab, iconlab: string[∗])

Sets the window and icon labels.

WinMapWindow (win: winWindow)
WinMapSubwindows (win: winWindow)
WinUnmapWindow (win: winWindow)
WinUnmapSubwindows (win: winWindow)

Maps or unmaps a window and/or all of its subwindows. Mapping a window onto the screen makes it and its subwindows visible; unmapping a window makes it and its subwindows invisible. Output to an unmapped window is allowed; when the window is remapped, its contents reflect such output.

WinMoveWindow (win: winWindow; pt: winPoint)

Moves the window to the given location relative to its parent.

WinEnableOutput (win: winWindow)
WinDisableOutput (win: winWindow)
WinUpdateWindow (win: winWindow)

Normally, output to an on a on-screen window is directed simultaneously to the window and to its backing pixmap. This can be disabled for performance reasons so that the output goes only to the pixmap; the window is then updated from the pixmap when **WinUpdateWindow**() is called.

WinFlush (win: winWindow)

Flushes all pending output for a window and its subwindows.

WinSync (win: winWindow; discard: bool)

Flushes the output buffer and waits for all requests to be received and processed by the X server. If *discard* is **true**, all pending window events not recognized by SRWin are discarded.

WinBell (win: winWindow; percent: int)

Rings the bell on the specified window, if possible. Volume is specified by the percentage relative to the base volume set by **xset**(1). *Percent* can be in the range −100 to 100 inclusive. If it is positive, the sound is louder than the base volume; if it is negative, the sound is quieter.

Cursors and Fonts

WinCreateCursor (win: winWindow; stdcursor: winStdCursor) returns cur: winCursor

Creates a standard cursor. Valid cursors are:

XC_X_cursor	*XC_arrow*	*XC_based_arrow_down*	*XC_based_arrow_up*
XC_boat	*XC_bogosity*	*XC_bottom_left_corner*	*XC_bottom_right_corner*
XC_bottom_side	*XC_bottom_tee*	*XC_box_spiral*	*XC_center_ptr*
XC_circle	*XC_clock*	*XC_coffee_mug*	*XC_cross*
XC_cross_reverse	*XC_crosshair*	*XC_diamond_cross*	*XC_dot*
XC_dotbox	*XC_double_arrow*	*XC_draft_large*	*XC_draft_small*
XC_draped_box	*XC_exchange*	*XC_fleur*	*XC_gobbler*
XC_gumby	*XC_hand1*	*XC_hand2*	*XC_heart*
XC_icon	*XC_iron_cross*	*XC_left_ptr*	*XC_left_side*
XC_left_tee	*XC_leftbutton*	*XC_ll_angle*	*XC_lr_angle*
XC_man	*XC_middlebutton*	*XC_mouse*	*XC_pencil*
XC_pirate	*XC_plus*	*XC_question_arrow*	*XC_right_ptr*
XC_right_side	*XC_right_tee*	*XC_rightbutton*	*XC_rtl_logo*

XC_sailboat	*XC_sb_down_arrow*	*XC_sb_h_double_arrow*	*XC_sb_left_arrow*
XC_sb_right_arrow	*XC_sb_up_arrow*	*XC_sb_v_double_arrow*	*XC_shuttle*
XC_sizing	*XC_spider*	*XC_spraycan*	*XC_star*
XC_target	*XC_tcross*	*XC_top_left_arrow*	*XC_top_left_corner*
XC_top_right_corner	*XC_top_side*	*XC_top_tee*	*XC_trek*
XC_ul_angle	*XC_umbrella*	*XC_ur_angle*	*XC_watch*
XC_xterm	*XC_None*		

WinSetCursor (win: winWindow; cursor: winCursor; fg, bg: winColor) returns c: winCursor

Sets the cursor of the specified window, returning **null** if unsuccessful. The cursor colors are set to *fg* (foreground) and *bg* (background).

WinFreeCursor (win: winWindow; cursor: winCursor)

Frees a cursor and reclaims any associated resources.

WinDefaultFont (win: winWindow) returns font: winFont

Returns the default font of the graphics context.

WinLoadFont (win: winWindow; fontname: string[∗]) returns font: winFont

Loads a font by name.

WinSetFont (win: winWindow; font: winFont)

Sets the font for the specified window.

WinFreeFont (win: winWindow; font: winFont)

Frees a font and reclaims any associated resources. The default font cannot be freed.

Clipping Manipulation Function

WinSetClipRectangles (win: winWindow; origin: winPoint; rects[∗]: winRectangle)

Sets the clipping region for a context window (clip rectangles are stored on a per-context basis). Subsequent output is clipped to be contained within the specified nonintersecting rectangles. The parameter *origin* is relative to the origin of the window, and the rectangle coordinates are relative to the *clip origin*.

Drawing Functions

WinClearArea (win: winWindow; area: winRectangle)

Clears a rectangular region using the *window* background color (which can differ from the current value set by **WinSetBackground**()). The clipping attributes of the context window are ignored.

WinEraseArea (win: winWindow; area: winRectangle)

Clears a rectangular area to the current graphics context background color, which is set using **WinSetBackground**().

WinCopyArea (srcw, destw: winWindow; src_rect: winRectangle; destp: winPoint)

Copies a rectangular region between (potentially) two windows on the same physical screen.

WinDrawArc (win: winWindow; box: winRectangle; a1, a2: int)
WinFillArc (win: winWindow; box: winRectangle; a1, a2: int)

Draws a (filled) arc, ellipse, or circle. The center of the circle or ellipse is the center of the rectangle. The major and minor axes are given by the width and height of the rectangle. The two angles are in units of degrees. The first angle specifies the start of the arc; the second specifies the path and extent of the arc, with positive values indicating a counterclockwise direction.

WinDrawLine (win: winWindow; pt1, pt2: winPoint)
WinDrawPolyline (win: winWindow; pts[∗]: winPoint)
WinDrawPolygon (win: winWindow; pts[∗]: winPoint)
WinFillPolygon (win: winWindow; pts[∗]: winPoint)

Draws a (filled) line, polyline, or polygon.

WinDrawPixel (win: winWindow; pt: winPoint)

Draws a pixel.

WinDrawRectangle (win: winWindow; rect: winRectangle)
WinFillRectangle (win: winWindow; rect: winRectangle)

Draws a (filled) box.

WinDrawString (win: winWindow; pt: winPoint; str: string[∗])
WinDrawImageString (win: winWindow; pt: winPoint; str: string[∗])

Draws a string. **WinDrawString**() alters only the pixels forming the characters of the text; **WinDrawImageString**() clears the ‘‘extent’’ of the text to the background color.

WinTextWidth (font: winFont; str: string[∗]) returns width: int

Computes the pixel width of a string in that font.

WinFontAscent (font: winFont) returns ascent: int
WinFontDescent (font: winFont) returns descent: int

Returns the ascent or descent of a font.

Drawing Attributes Manipulation Functions

WinSetLineAttr (win: winWindow; line_width: int; l: winLineStyle; c: winCapStyle; j: winJoinStyle)
WinSetFillAttr (win: winWindow; fill_style: winFillStyle; fill_rule: winFillRule)
WinSetDashes (win: winWindow; dash_offset: int; dash_list: string[∗])
WinSetArcMode (win: winWindow; arc_mode: winArcMode)
WinSetDrawOp (win: winWindow; dop: winDrawOp)

Sets the line drawing characteristics of a window. Constants include:

winLineStyle:	*LineSolid, LineDoubleDash, LineOnOffDash*
winCapStyle:	*CapNotLast, CapButt, CapRound, CapProjecting*
winJoinStyle:	*JoinMiter, JoinRound, JoinBevel*
winFillStyle:	*FillSolid, FillTiled, FillOpaqueStippled, FillStippled*
winFillRule:	*FillEvenOddRule, FillWindingRule*
winArcMode:	*ArcChord, ArcPieSlice*

winDrawOp:

Op_Clear, Op_And, Op_AndReverse, Op_Copy, Op_AndInverted, Op_Noop, Op_Xor, Op_Or, Op_Nor, Op_Equiv, Op_Invert, Op_OrReverse, Op_CopyInverted, Op_OrInverted, Op_Nand, Op_Set

The defaults are line width 0, *LineSolid*, *CapButt*, *JoinMiter*, *FillSolid*, *FillEvenOddRule*, *ArcPieSlice*, *Op_Copy*, and no dashes.

Using a line width other than zero may degrade performance on some X servers.

Drawing operations other than *Op_Copy* are potentially nonportable or even undefined and should be used only with a clear understanding of the X color model. For example, *Op_Xor* gives different results (other things being equal) on Sun and DEC hardware.

WinSetForeground (win: winWindow; foreground: winColor) returns pv: winPixel
WinSetBackground (win: winWindow; background: winColor) returns pv: winPixel

WinSetForegroundByPixel (win: winWindow; foreground: winPixel)
WinSetBackgroundByPixel (win: winWindow; background: winPixel)

Sets the foreground or background color to be used in subsequent drawing operations.

Event Handling Functions

WinSetPoll (win: winWindow; msec: int)

Sets the interval between event checks, in milliseconds. The default interval is 100 milliseconds.

WinSetEventMask (win: winWindow; em: int)

Registers events of interest for a window.

Valid event masks are the same as the event types. They can be *or*'ed together to set multiple masks for a window. The default event masks include all supported event types if an event channel is provided at the window creation time, or nothing if no channel is provided and the window is requested to be mapped.

There are several defined events masks:

Ev_KeyDown	*Ev_EnterWindow*	*Ev_All*
Ev_KeyUp	*Ev_PointerMove*	*Ev_None*
Ev_ButtonDown	*Ev_ExitWindow*	
Ev_ButtonUp	*Ev_DeleteWindow*	

Ev_DeleteWindow is sent whenever the window manager issues a DELETE_WINDOW message (e.g. the user chooses *Quit* from window manager's menu); this event can only be received in the top-level window. *Ev_All* is the combination of all possible events; *Ev_None* selects no events.

Image Manipulation Functions

WinCreateImage (win: winWindow; depth, w, h: int) returns im: winImage

Creates an image with width *w* and height *h*. If **UseDefault** is passed as *depth*, the depth of the image is set to be the same as that of the physical display.

WinDestroyImage (im: winImage)

Destroys the image and frees the memory space it occupies.

WinGetPixel (im: winImage; pt: winPoint) returns pv: winPixel
WinPutPixel (im: winImage; pt: winPoint; pv: winPixel)

Reads or writes a pixel value from or to the image. The point must be inside the image. These two functions are not protected by mutual exclusion; the application program should be aware of potential consistency problems when an image is shared among multiple processes. **WinPutPixel**() is most reliable when different processes work on different rows (and therefore different memory words).

WinAddPixel (im: winImage; pv: winPixel)

Increments each pixel in the image by the value of *pv*. This function is not protected by mutual exclusion.

WinGetImage (win: winWindow; im: winImage; src_rect: winRectangle; dest: winPoint)
WinPutImage (win: winWindow; im: winImage; src_rect: winRectangle; dest: winPoint)

Copies the rectangular area specified by *src_rect* on the window to the image starting at point specified by *dest*, or copies an image to a window. The depth of the image and the window must match.

SEE ALSO

sr(1), srl(1)

Qiang Alex Zhao, *SRWin: A Graphics Library for SR.* TR 93-14, Dept. of Computer Science, The University of Arizona, 1993. Included in the SR distribution.

Adrian Nye, *Xlib Programming Manual, Volume One, 3rd ed.* O'Reilly & Associates, Inc., 1992, ISBN 1-56592-002-3.

Adrian Nye, *Xlib Reference Manual, Volume Two, 3rd ed.* O'Reilly & Associates, Inc., 1992, ISBN 1-56592-006-6.

Robert W. Scheifler and James Gettys, *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD, 3rd ed.* Digital Press, 1992, ISBN 1-55558-088-2.

CAVEATS

Numerous unexplained problems have been seen on the Sequent Symmetry.

Except on an SGI Iris, SRWin does not function properly if MultiSR is enabled.

AUTHOR

Qiang A. Zhao.

ACKNOWLEDGMENTS

SRWin was inspired by X-Icon, an X interface developed by Clint Jeffery for the Icon language. Many useful ideas came from the *Winpack* graphics library created by Scott Hudson.

NAME

srmap − network mapping file for SR

DESCRIPTION

Srmap is used to find the executable program when a virtual machine is created with an ‘**on** *n*’ clause and an explicit pathname has not been given for machine *n.*

Srmap is read from a standard location; this may be overridden by supplying the name of a file in the environment variable *SRMAP.*

Lines in the file have two whitespace-separated fields, a *pattern* and a *template.* Lines with less than two fields are ignored. Comments are introduced by ‘#’ and terminate at end of line.

When an SR program runs, a program *location* is defined by concatenating the hostname, a colon (:), and the absolute pathname of the executable. The hostname is chopped at the first ‘.’, if any. The location is matched against each pattern in turn until one succeeds; then the corresponding template is used to form a network pathname. This pathname is passed to *rsh*(1) on the remote machine.

Patterns have the general form *host:path,* with these characters having special meaning:

?	matches any single character except ‘/’
*	matches any string of characters except ‘/’
**	matches any string of characters *including* ‘/’

Templates construct network file paths corresponding to matched patterns. Each occurrence of **$***n* in a template is replaced by the characters matched by the *n*th ?, *, or ** in the corresponding pattern. *n* may be 1 to 9, or A to Z for the 10th through 35th strings.

EXAMPLE

Consider the mapping file:

```
# sample mapping file
*:/r/**        /r/$2            # already a network path
client:/**     /r/server/$1     # client disks are on server
???*:/**       /r/$1$2$3/$5     # general rule for others
```

A program run on host ‘caslon.arizona.edu’ generates these different network paths, depending on the program’s path:

local path	*network path generated*
/r/bas/usr/abc/foo	/r/bas/usr/abc/foo
/usr/xyz/bar	/r/cas/usr/xyz/bar

Note that ‘caslon:’ was prepended to the program path before scanning the patterns.

CAVEATS

Srmap is designed for environments providing transparent access to remote disks via a systematic naming scheme. Other environments may require explicit path specification by the SR program.

NAME

srtrace − runtime event trace file for SR program

DESCRIPTION

The runtime events of any SR program can be recorded during execution in an *srtrace* file by setting the environment variable SR_TRACE to the name of the *srtrace* file. Two specific names, *stdout* and *stderr*, can be used to direct the trace output to standard output and standard error output respectively.

Each line of the *srtrace* file has the following fields:

Filename

Name of the file where the corresponding SR statement is located.

Line number

Line number of the corresponding SR statement.

Proc name

Proc name of the corresponding SR statement. This has the form

[*vm*(*n*).]*resource*.*proc*

which reads as *proc* in *resource* of virtual machine *n. vm(n)* is displayed only for distributed programs.

Event

Name of the event used in *srtrace* file. The available events are as follows:

Event	**Meaning**	**SR statement**
CREATER	creation of resource	create
CREATEG	creation of global	import
CREATEV	creation of virtual machine	create
DESTROYR	destruction of resource	destroy
DESTROYV	destruction of virtual machine	destroy
CALL	synchronous invocation	call
SEND	asynchronous invocation	send
FORWARD	transfer of responsibility for reply	forward
REPLY	reply and continue execution	reply
RETURN	terminate and return	return
BODY	beginning of resource init code	resource
ENDBODY	end of resource init code	end
FINAL	beginning of resource final code	final
ENDFINAL	end of resource final code	end
PROC	beginning of service by proc	proc
ENDPROC	end of service by proc	end
IN	entry to input statement	in
ARM	service of arm of an input statement	–>
NI	exit from an input statement	ni
CREATES	creation of semaphore	sem
INITS	initialization of semaphore	
P	beginning of P operation	P
CONTP	completion of P operation	
V	V operation	V
CO	beginning of co statement	co
OC	end of co statement	oc

Process ID

A hexadecimal number identifying the particular process that generated the event.

Additional Field

For the following events, there is a second hexadecimal number following the process ID. The meanings are:

RETURN	process ID of the invoker
REPLY	process ID of the invoker
FORWARD	process ID of the invoker
ARM	process ID of the invoker
NI	process ID of the invoker
PROC	process ID of the invoker
ENDPROC	process ID of the invoker
INITS	initial value of the semaphore
CREATES	semaphore ID
P	semaphore ID
CONTP	semaphore ID
V	semaphore ID

Other events have 0 in the place of this field.

EXAMPLE

Some sample output:

CS.sr, 15	main.body	BODY	1730b8	0
CS.sr, 16	main.body	CREATEG	1730b8	0
CS.sr, 1	CS.body	BODY	173168	0
CS.sr, 5	CS.body	SEND	173168	0
CS.sr, 1	CS.body	ENDBODY	173168	0
CS.sr, 5	CS.arbitrator	PROC	1731c0	173168
CS.sr, 20	main.body	SEND	1730b8	0
CS.sr, 15	main.body	ENDBODY	1730b8	0
CS.sr, 7	CS.arbitrator	IN	1731c0	0
CS.sr, 20	main.user	PROC	173168	1730b8
CS.sr, 22	main.user	CALL	173168	0
CS.sr, 8	CS.arbitrator	ARM	1731c0	173168
CS.sr, 7	CS.arbitrator	NI	1731c0	173168
CS.sr, 10	CS.arbitrator	IN	1731c0	0
CS.sr, 24	main.user	SEND	173168	0
CS.sr, 10	CS.arbitrator	ARM	1731c0	173168
CS.sr, 10	CS.arbitrator	NI	1731c0	173168
CS.sr, 7	CS.arbitrator	IN	1731c0	0
CS.sr, 20	main.user	ENDPROC	173168	1730b8

SEE ALSO

sr(1), srprof(1)

CAVEATS

srtrace output reflects the actual SR implementation, which differs in some details from the SR source language. For example, a *process* statement is traced as a SEND followed by a PROC. Some *P* and *V* statements are implemented as *in* and *send* respectively, and vice versa.

The following *srtrace* events report a line number different from the line number of the corresponding SR statement:

NI	line number of corresponding IN
ENDPROC	line number of corresponding PROC
ENDBODY	line number of corresponding BODY
ENDFINAL	line number of corresponding FINAL