Section 1 -- Introduction

Section 1 - Introduction

UNISEX, a **UNI**x-based **S**ymbolic **EX**ecutor for Pascal, provides an environment for testing and verification of programs. This document is a guide to using UNISEX. Section 2 presents an overview of the capabilities and usage of the symbolic executor itself. Section 3 describes UNISEX Pascal, a Pascal subset with an integrated assertion language. A more detailed description can be found in The UNISEX Pascal Language Reference Manual. Section 4 is a detailed listing of the symbolic executor commands. The symbolic executor has been developed to run on the VAX 11/780, running under the Unix operating system. It also requires the use of the Franz Lisp interpreter environment. UNISEX consists of a cross compiler and a set of Lisp utility routines. The cross compiler was developed using the Lex lexical analyzer generator and the YACC compiler generator, and takes as source code the Pascal program to be symbolically executed. The output of the compiler is a valid Franz Lisp source program. This Lisp program, when run under the Franz Lisp interpreter (along with the utility routines) is an interactive symbolic executor of the original Pascal program.

Section 2 -- UNISEX System Features

Section 2 - UNISEX System Features

The UNISEX system provides a symbolic execution approach for testing or verification. Its key features are:

It supports a large subset of Pascal, described in the UNISEX Pascal Language Reference Manual.

It runs on UNIX, and is thus easier to use than similar systems on less friendly operating systems.

It allows the user to choose test or verify mode. In test mode, it gives the user the option of executing with numeric or symbolic values. In verify mode, it gives the user the option of automatic or manual execution.

Overview of Operation Overview of Operation The UNISEX system is currently invoked with a shell script in unisex (/usr/local/unisex on the UCSB Computer Science Instructional Vax). A Pascal program in the file *program.p* would be executed with the command: **unisex** *program.p* This causes the Pascal program to be compiled and loaded, after which the user is prompted to choose test or verify mode and initialize the UNISEX debug functions. In test mode UNISEX will prompt the user to initialize global variables. The user may then enter a sequence of assignments of the form *variable = value*, separated by commas. If all of the values are numeric UNISEX will act like a symbolic debugger. UNISEX can be used for generating path predicates or for more general testing by giving variables symbolic values.[1] [2] Symbolic values entered by the user must have the same syntax as Pascal identifiers. UNISEX-generated symbolic values will generally begin with a special symbol: for scalar values, and for pointer values. In this mode the user has available all of the UNISEX commands presented in Section 4 except **verify** and **autoverify**. In verify mode all variables are assigned symbolic values by the system, and the full command set is available. In this mode UNISEX keeps track of verified paths, as well as paths that have been traversed but were not verifiable.[3] [4] Since the UNISEX simplifier is not very powerful, the user is asked to evaluate most verification conditions and branch conditions. UNISEX actually has two simplifiers; one for programs with variables of type *real* and the other for programs without variables of type *real*. Internally the difference is that the *real* simplifier does not use all of the rewrite rules used by the *integer* simplifier. From the user's point of view the difference is that two identical expressions may not look the same after simplification if one occurs in a program with real numbers and the other in a program without real numbers. These can be reviewed with the **paths** command.

Trace options Trace options UNISEX allows the user to choose the amount of information displayed as execution progresses. Turning the *branch* option on causes all decisions to be displayed. Turning *verb* on does that, and also displays statements just before they get executed, and the results of assignments. The default for both is off. In addition, the simplifier may be turned off so that the results of all operations will be stored and displayed in unsimplified symbolic form.

Breakpoints Breakpoints UNISEX allows the user to set and remove breakpoints at any statement in a program at any time (except in autoverify mode, when breakpoints are ignored). In addition the user may set *step* on, which is equivalent to having a breakpoint at every statement.

States States The of a UNISEX symbolic execution consists of a path condition (*pc*), a program counter and the current value of each variable. The *pc* is a predicate that represents all of the known constraints on symbolic values occurring in the program. It is always initialized to before starting an execution, which says that nothing is known about any symbolic values. Whenever a decision is unresolvable (i.e., *pc* is not strong enough to imply the test or its negation), both branches must be traversed to completely cover the execution tree [2]. To do this, it must be possible to postpone one of the branches while continuing the other. In UNISEX, the user is given almost unlimited freedom in moving around in the execution tree, through the use of the **save** and **restore** commands. The state can be saved at any time, and any saved state can be restored at any time, without affecting the other saved states. This allows the user to, for example, choose a branch, pursue it for a while, saving more states, and then decide that it would be better to do the other branch first instead. This other branch could be restarted with the **restore** command without affecting the subsequently saved states.

Miscellaneous Displays Miscellaneous Displays At any time UNISEX is displaying the prompt the user may look at: fragments of the program, the path condition, a list of the saved states, the values of variables, and a few more things detailed in Section 4.

Test Mode Test Mode Any valid UNISEX Pascal program may be executed in test mode. Assertion language statements may be included, but are not necessary. Details of what each assertion statement type does can be found in Section 13 of the Language Reference Manual. In this mode the path condition can be modified in four ways: by encountering an **entry** or **assume** statement, by the user explicitly modifying it with the **addpred** command, or by executing an unresolvable decision statement. As noted in the overview, UNISEX prompts the user to initialize global variables in test mode. Variables may be given either integer or symbolic values at this time. All and only those variables initialized in response to this prompt will have initial values accessible by using the initial value operator. Therefore, the user need only use this means of initializing variables if there are assertion language statements in the program that contain variable references. For all other initialization needs the standard Pascal procedure *read* is probably more convenient. The first two programs in the Appendix demonstrate the use of test mode.

Verify Mode Verify Mode To be executed in this mode a program must be minimally asserted. That is, the program must have a **prove** or **exit** assertion, all loops must have associated **assert** statements, all labels (i.e., potential targets of goto statements) must label **assert** statements, and all subroutines must have **exit** assertions. A subroutine without an **exit** assertion causes a UNISEX run-time error. Failure to meet any of the other conditions will make it impossible to verify the program, but will not signal an error. UNISEX allows either manual or automatic traversal of the execution tree in verify mode. When a program is verified manually, the user is responsible for saving states as necessary, choosing branches at unresolvable decision points, and restoring states for continuing postponed branches. In automatic mode, this is all done by the system. UNISEX does a depth first traversal of the tree, always stacking the branch and continuing the branch at forking decision statements. The other difference between the two modes is that in automatic mode the user is not allowed to set breakpoints, save or restore states, or change variable values. When verifying programs with subroutines there are two steps that are not necessary for programs without subroutines. First, it must be shown that the subroutine is consistent with its entry and exit assertions. Second, when verifying the program, each time the subroutine is called its entry assertion must be shown to hold and its exit assertion is assumed to be (i.e., the exit assertion is evaluated and conjoined to *pc*). The first is done in UNISEX with the **verify** *routine_name* command for a manual verification, and with the **autoverify** *routine_name* command for semi-automatic verification. The second is done in manual mode as default, or automatically with the **autoverify program** command. Subroutine calls are verified as follows. First, the verification condition *pc* **implies** subroutine_entry_assertion is generated (and sent to the Theorem Prover). Next, all **var** parameters of the subroutine are given fresh symbolic values. Finally, the subroutine exit assertion is evaluated using these fresh values and conjoined to *pc*. This conjunction represents the effect of the subroutine, which is why exit assertions are required for all subroutines. A subroutine body is verified by treating it as a program.[5] [6] If a subroutine uses any global user-defined types, then the declarations of those types must have been before the subroutine body can be verified. This occurs because UNISEX does not have any knowledge of types or variables until their declarations have been processed.

Thus it may be necessary to, for example, put a breakpoint somewhere after the global declarations, run the program (with the **go** command) to the breakpoint, and then issue the **verify** command. In doing this, it is assumed that the subroutine has no side-effects and that no aliasing of **var** parameters occurs. It the user's responsibility to check that these assumptions are valid. In automatic mode all possible paths of the subroutine will be traversed before control returns to the program level. The user has no control over this, except by using the **quit** or **restart** commands. In manual mode, the user may choose to traverse some subset of the possible paths. Whenever control reaches the end of the routine block, the user is warned that control will return to program level (i.e., the point at which the **verify** command was issued) if a state is not restored. At this time the user may either restore a state to continue another path, or issue the **go** command, which will return.

Subroutine calls in a subroutine are handled the same way as in the main program block. **Verify** and **autoverify** may be used for routines nested arbitrarily deeply, but the commands themselves *cannot* be nested. That is, if a **verify** command has been issued, then neither **verify** nor **autoverify** may be used until control returns to the program level. The final program in the Appendix demonstrates the use of verify mode.

Section 3 -- UNISEX Pascal Summary

Section 3 - UNISEX Pascal Summary The language of the programs to be symbolically executed is a subset of standard Pascal, along with an assertion language. A detailed description of the differences between the subset and standard Pascal is given in the UNISEX Language Reference Manual. A summary of the Pascal subset and the assertion language follows. Programs may contain the following data types:

1) integer
2) boolean
3) real
4) char
5) arrays
6) records
7) pointers
8) sets
9) user-defined types

The valid Pascal statement types are the following:

1) declarations
   a) labels
   b) constants
   c) types
   d) variables
   e) procedures and functions
2) assignments
3) if...then...
4) if...then...else...
5) while...do...
6) repeat...until...
7) for...to...do...
8) for...downto...do
9) case statements
10) goto statements

The major features included in standard Pascal but not included in the symbolically executed language are the following:

1) file types
2) scalar ordering
3) variant records
4) most standard procedures and functions The assertion language consists of six types: **entry**, **exit**, **assume**, **prove**, **assert**, **axiom**; four meta-symbols: and three additional keywords: **forall**, **exists**, **implies**. Assertion syntax is: {: **keyword** (*predicate_list*) :} where **keyword** is one of the six assertion types and *predicate_list* is a list of valid UNISEX expressions separated by commas. If entry and/or exit assertions appear in a program or procedure, they must be placed after variable declarations and before the **begin**. Zero or more axioms may appear after entry and exit assertions and immediately before the **begin**. **Assume**, **prove**, and **assert** are to be used like Pascal statements, which means that they are separated by semicolons the same way. Note that the semicolon is placed the metacharacter **Entry** and **exit** assertions and **axioms**, on the other hand, are to be treated as statements; they appear only in the declaration section, and are thus never followed by semicolons.

Examples: {: **assume** ((x < y),(y <> 0)) :}; {: **prove** (z=y'*x',x = 5) :}; {: **assert** (y<z, z<w) :}; This form allows programs to be compiled by a standard Pascal compiler, even when they contain these special state-

ments. The standard compiler treats a statement of this form as a comment (possibly) followed by an empty statement. The additional UNISEX keywords, **forall**, **exists** and **implies**, represent universal quantification, existential quantification, and logical implication respectively. They may be used only within the and delimiters, as may the prime, used to represent a variable's initial value, and used to indicate the occurrence of an uninterpreted function reference.

Section 4 -- Command Summary

Section 4 - Command Summary

Whenever the UNISEX prompt [8] The prompt indicates that UNISEX is in command interpreter mode. Another prompt, is displayed when the user is requested to initialize global variables. At that time only a sequence of comma-separated assignments, or a carriage return, will be accepted. is displayed, the system is waiting for an input. In response the user may execute the commands listed below. Upon receiving a command in response to the prompt, the system will execute the command and return again with the prompt. This will continue until either the **go** command is issued or one of the expected responses is received. The user may also enter Pascal-style comments; these will be ignored by UNISEX. In what follows words in **boldface** are to be typed exactly as they appear, and words in *italics* represent parameters to be supplied by the user.

The Executor options and their definitions are:

Continuing Execution

go

This command forces the executor to leave the routine that interacts with the user, and to continue execution of the program.

quit

This command terminates the execution in progress and prompts the user to return to unix or start a new execution.

restart

This command causes the current state to be erased, and the current program to be restarted before the "test or verify" prompt.

Displays to User

list
list *n*
list *n1*,*n1*

The first form always lists the entire Pascal program, even if the current scope is a subroutine, as when a **verify** command has been issued. The second lists line *n* if it exists. The third form lists the range of lines between *n1* and *n2* inclusive. The line numbers refer to the UNISEX pretty-printed form, and do not necessarily correspond to line numbers in the source text.

curbr

Current Breakpoints. This command displays the line numbers on which there are currently breakpoints.

pc

Path Condition. Displays the currently known program constraints.

states

Displays a list of the currently stored states. For each saved state the display includes the number of the state, the line number at which the state was saved, the routine in which that line appears, and any comment the user included with the **save** command.

types

Displays a list of the *local* user defined types.

vars

Displays a list of the *local simple* variables and their values, as well as the current value of the path condition.

var *varname*

Displays the value of variable *varname*. This command allows the value of any variable known in the current scope to be displayed, including elements of arrays and fields of records, and non-local variables.

paths

In test mode this command displays a list of the statements (by line number) that have been executed (the
In verify mode it displays a list of verified paths, a list of traversed but un-verified paths, and the current path.

axioms

Displays the axioms appearing in the current program.

help

Displays a list of the commands available within UNISEX.

Saving and Restoring States

save *comment*

Saves the current state of the symbolic execution. The comment is whatever is typed after the keyword **save**, up to the next newline. The system displays a confirmation that includes the comment and a system-assigned state number. This state number is used for restoring and removing states.

restore *n*

Restores state *n* to be the current state if it exists. If it doesn't exist, a message to that effect is displayed. The numbers (and user-supplied comments) of the currently saved states can be viewed using the **states** command described above.

rmstate *n*

Removes state *n* from the list of saved states, if it exists.

Breakpoints and Program Stepping

brk off
brk on

brk off disables the breaking option, so that even if there are breakpoints set, the system will ignore them. The breakpoints are not removed, however. This allows the user to continue uninterrupted execution of the program without having to use the **rembr** command to remove all the breakpoints.
brk on enables the breaking option after it has been turned off by the above command. The default is *on*.

setbr *n1,n2,n3,...*

The **setbr** command sets breakpoints at line numbers *n1, n2, n3, ...* in the program (with reference to the pretty-printed Pascal source code). For example: # setbr 3,7,15 sets breakpoints at lines 3, 7 and 15. During program execution, the program will halt *before* execution of any line with a breakpoint set. This command automatically enables the breakpoint option.

rmbr *n1,n2,n3, ...*

Removes breakpoints from any of lines *n1, n2, n3, ...* of the program that currently have a breakpoint set.

step on
step off

The step option halts the program at each new line of the program. This is equivalent to having a breakpoint on each line of the program. This allows the user to step through the program. Execution is continued with the **go** command. The default is *off*.

Verification Commands

verify *name*

This command is used to verify subroutines. The system must be in verify mode and *name* must be an existing subroutine. If these conditions are not met, then an error message is printed and the command ignored. Otherwise this command has the following effect:

First the current state is stored, so that when the subroutine verification is completed (or abandoned), everything will be as if the command was never issued, except that any verified and un-verified paths in the routine will have been added to the appropriate lists for viewing with the **paths** command. The requested routine temporarily becomes the top level. Now the user may execute the subroutine as if it were entered as a program, with the following exception: neither **verify** nor **autoverify** may be used until the current call has returned.

autoverify program

This command causes the program to be verified which means that unresolvable decisions are saved and restored as necessary without user intervention.[9] [10] Except in the role of theorem prover. Execution will proceed until all possible paths have been traversed. Several commands become inoperative when autoverify is selected. These are: change, step, brk, setbr, rembr, curbr, save, restore, rmstate, autoverify, verify.

autoverify *name*

This command is equivalent to the verify command above, but saves and restores unresolvable decision points automatically. When all paths in the subroutine have been traversed, control returns to the point at which the autoverify command was issued.

Running Trace Options

branch on
branch off

With the *branch* option on, the program prints the evaluated result of each decision point, even when the Path Condition is strong enough to make the decision without user assistance. The default value is *off*.

verb on
verb off

With the *verbose* option on, the program displays each program line just before the line is to be executed. It also prints out the evaluated result of each decision point it encounters, as in branch mode above, and the result of each assignment statement. The default value is *off*.

Miscellaneous Options

change *varname* = *newval*

This command changes the value of variable *varname* to *newval*. *Newval* must be an integer or symbolic value. For example: # change arr[1,5] = valueX2

addpred *expression*

This command allows the user to add a predicate to the current Path Condition.[11] [12] This is sometimes useful for building loop invariants on the fly, but should otherwise be avoided, since it will usually cause the path condition to be inconsistent with the program being executed. Only relations of the form may be added, where *v1* and *v2* are integer or symbolic values, and **relop** is a Pascal relational operator.

simplify on
simplify off

If the simplifier is turned off, all expressions will be left in unreduced form, even if they contain only numbers. The default is *on*.

Bibliography

Bibliography

[1] Foderaro, John K., Regents of the University of California, 1981

[2] Hantler, Sidney L., and James C. King, *Computing Surveys,* Vol. 8, No. 3, Sept. 1976.

[3] Jensen, K., and N. Wirth, Second Edition, Springer-Verlag, 1974

[4] Johnson, Stephen C., Bell Laboratories, Murray Hill, New Jersey, 1978

[5] King, James C., *Comm. ACM*, Vol. 19, No. 7, July 1976.

[6] King, J.C., Chibib, A.C., Hantler, S.L., Version 8.4, May 1978.

[7] Lesk, M. E., and E. Schmidt, Bell Laboratories, Murray Hill, New Jersey

Appendix - Example UNISEX sessions

Appendix - Example UNISEX sessions

A User’s Manual for the UNISEX System

Table of Contents