**Appendix-A: Tool Demo**

To illustrate the features of the static debugger and how they help debugging, we provide a simple example with three different procedures. There are different types of errors in the procedures.

1. Defined Predicates and Procedures

The program defines three predicates as follows,

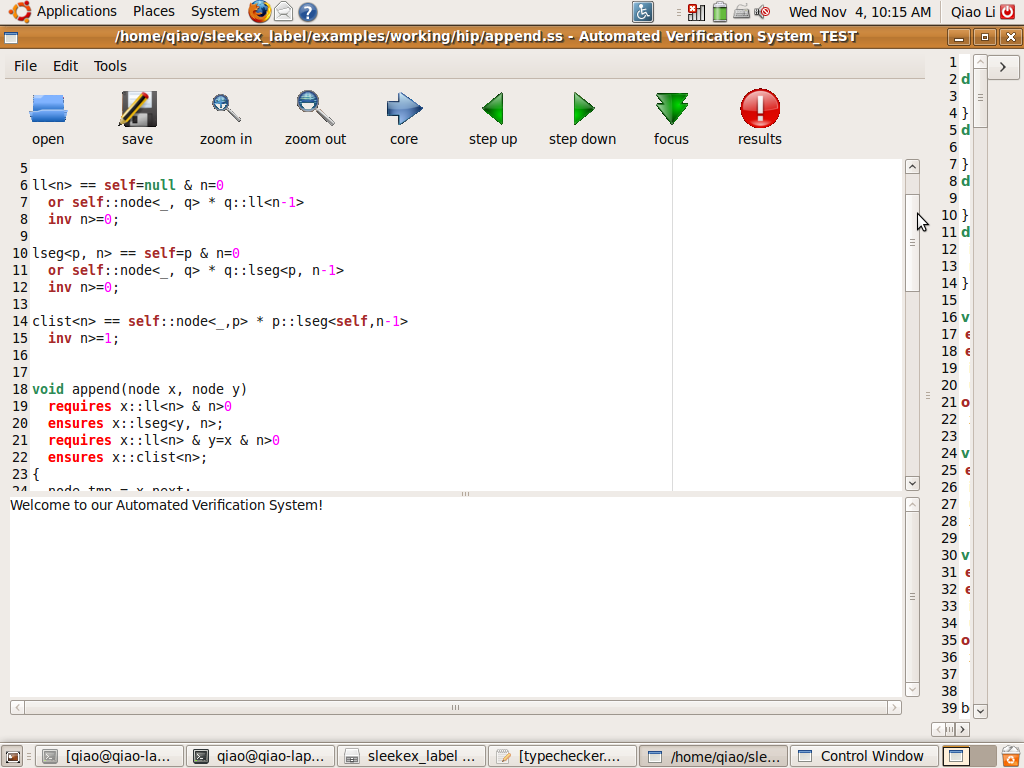
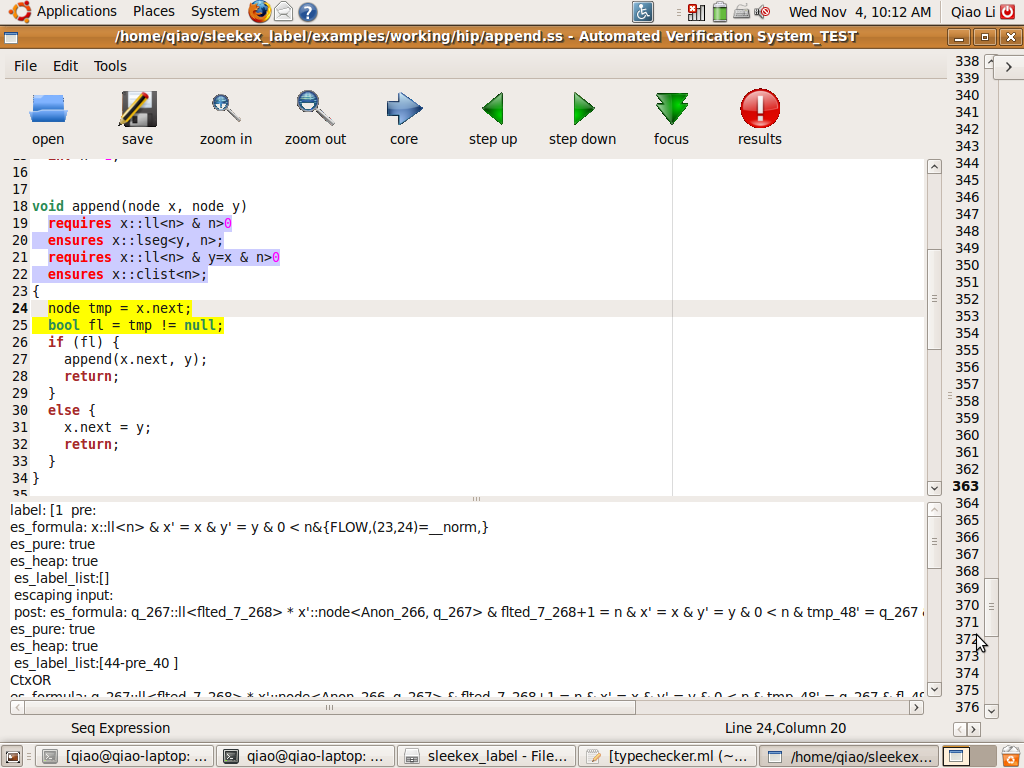
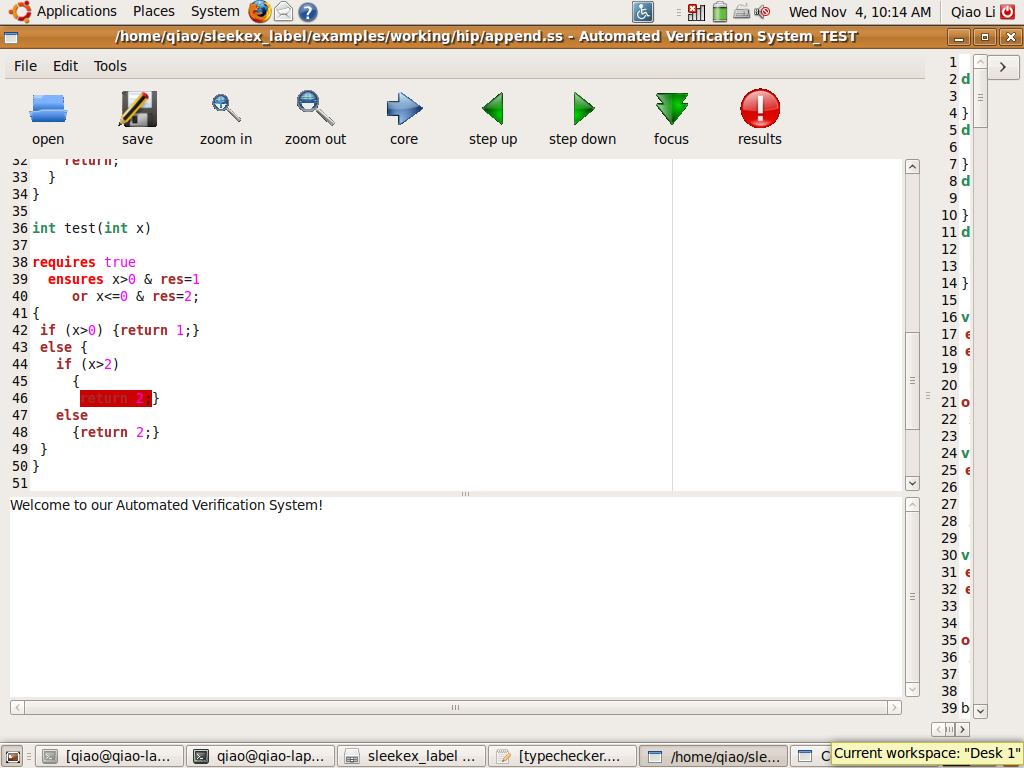


Figure 11. Sample predicates

ll<n> defines a basic linked list that can be empty. lseq<p,n> is a more restricted linked list that p has to be the last node of the linked list. clist<n> is a cyclic linked list with at least one node (n>=1).

The program includes the following procedures,





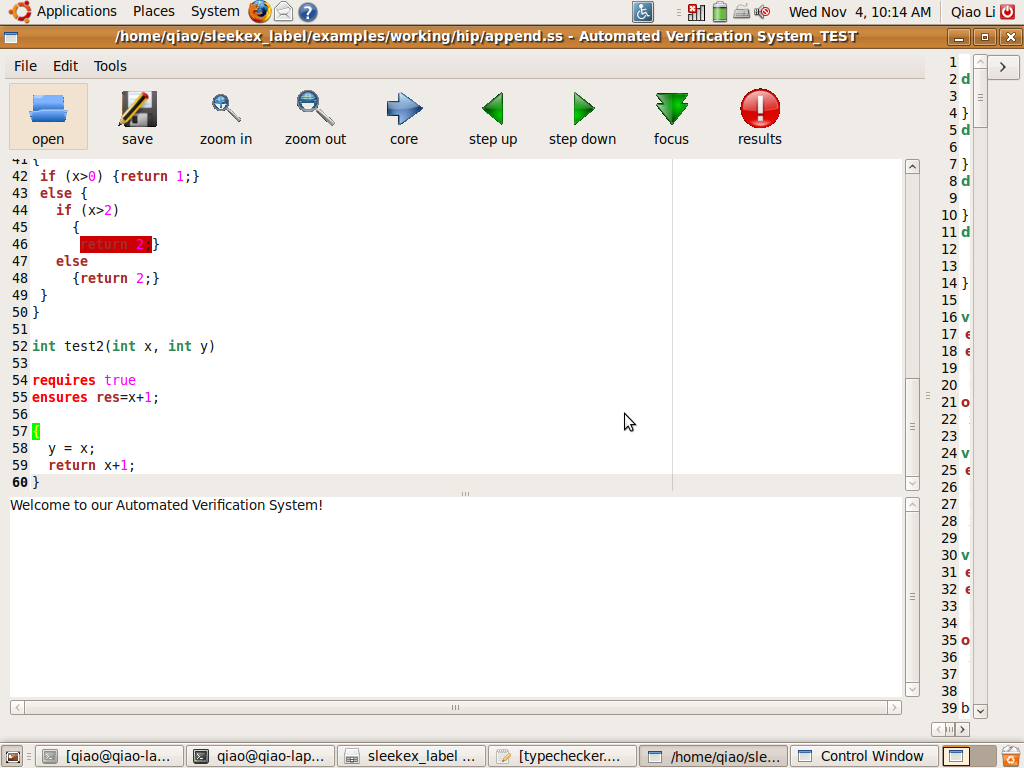


Figure 12. Sample Procedures

The first procedure ($append) is the basic append function that append the node y to the linked list x. The other two procedures ($test, $test2) are using conditions to return different values, created for demonstration purposes.

When the user presses the “open” button on the left, the user can select this particular file from the dialogue.

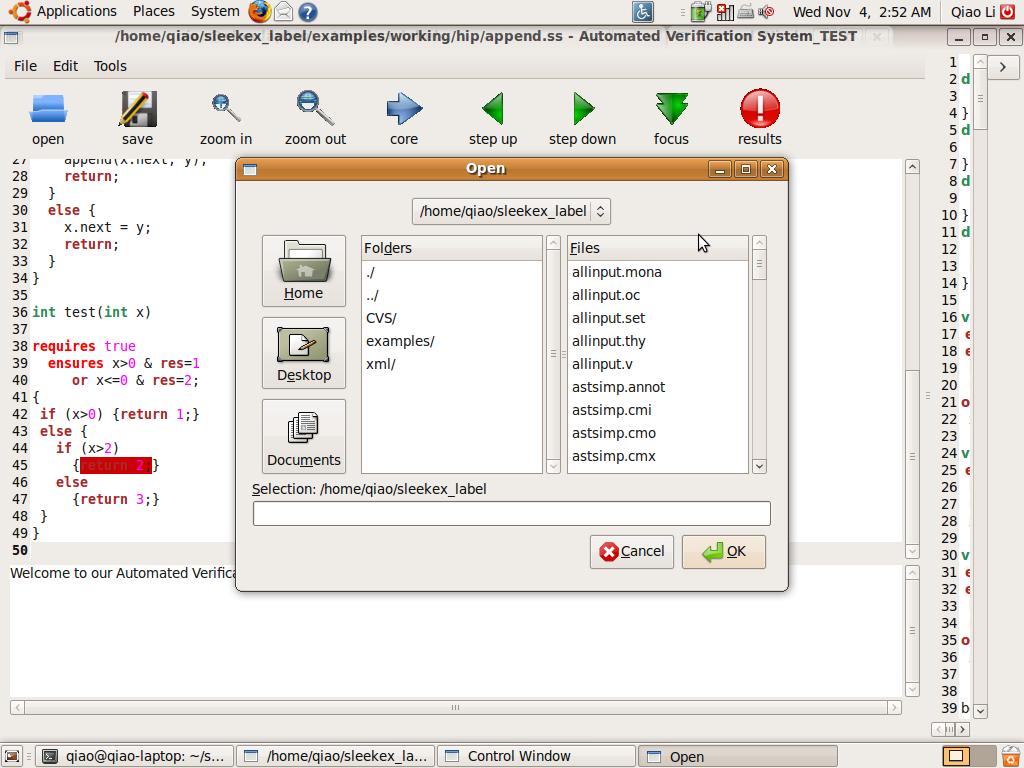


Figure 13. Open Dialogue

Once the file is open, we get the following verification results in Figure 13.

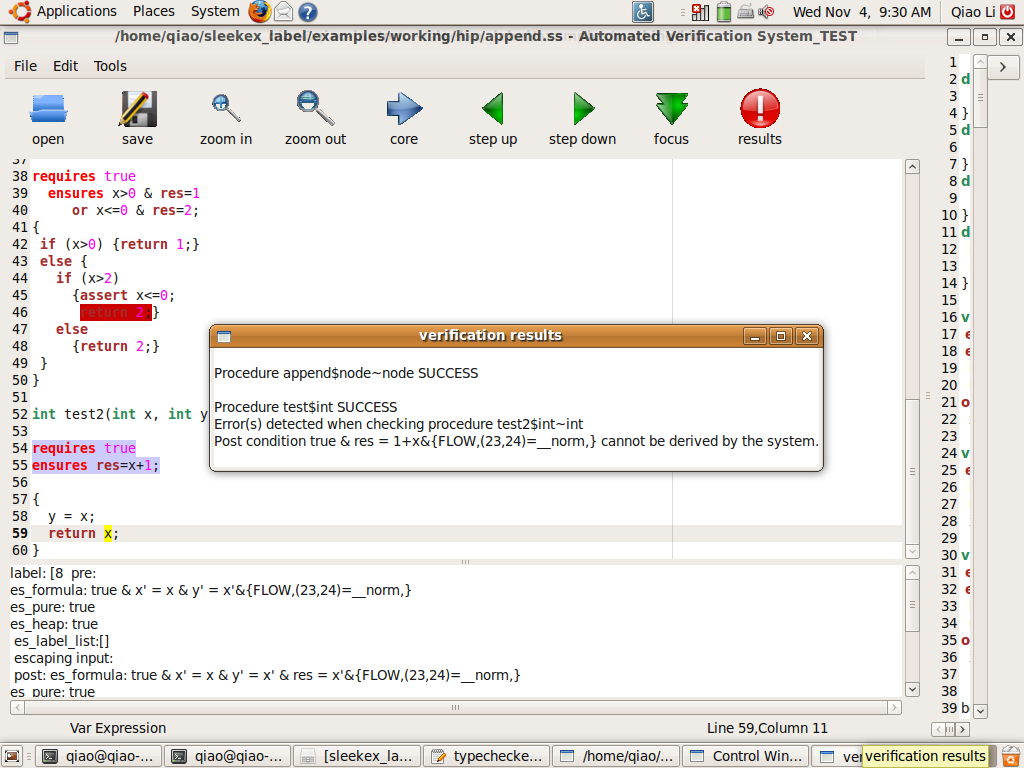


Figure 14. Verification Results Window

1. False Post Condition

Firstly, the results indicate that the postcondition of $test2 cannot be fulfilled in Figure 13. The user can jump to the last line of $test2 to obtain the program states of the last line. The state shows that the result will be x, which is not the defined postcondition (x+1).

If we change the last line from “return x;” to “return “x+1;” and press “save” button, the verification system will be rerun again. This time, the verification results show that $test2 has passed the verification test.

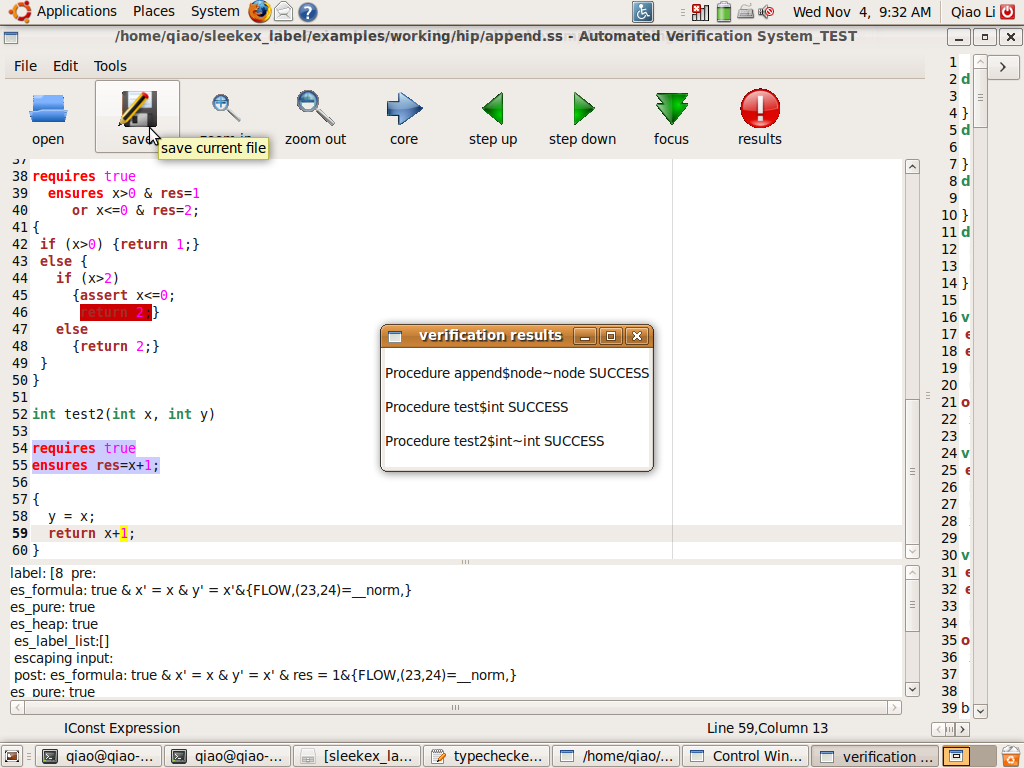


Figure 15. Verification Results after Debugging

1. Dead codes

Also, we can see that part of the codes in $test has been highlighted red. This initial red highlight means that part of the codes is dead code, which is unreachable in the normal flow of the program. The system indicates this procedure fulfills the stated specification. However, dead code means there are logic flow and we wish to eliminate that. Then, we click on the line above the dead code, we can observe the program states just before the execution in Figure 15.

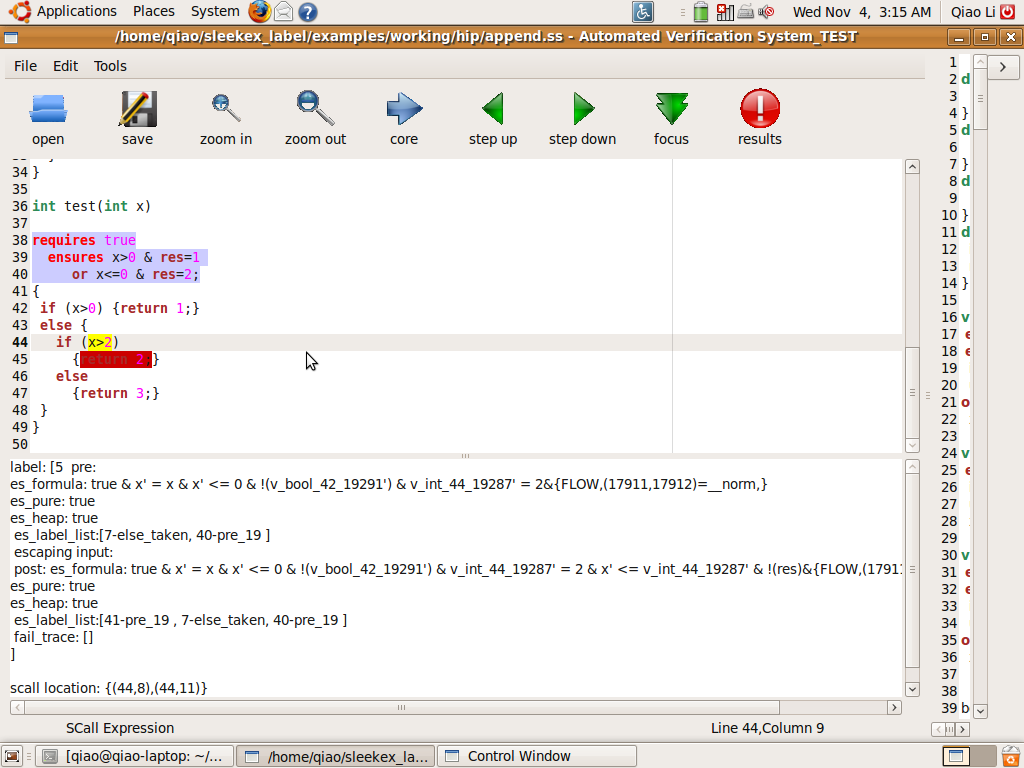


Figure 16. Function of Heap States

The part that “!res” shows no result at this statement. We can see that there are two Boolean values, “x’<= 0” and “x’ <=2”. This not as expected, because from the sub-expression “x>2”, we do not want “x’<=2”. The sequence of the values in the program state represents the sequence that these values have been derived. So if we already have “x’<=0”, we will not be able to get a state of “x>2”, as expected by this highlighted sub-expression. Hence, we have to change this Boolean condition in order to eliminate the dead codes.

1. “Assume” command

If we add in an “assume ‘x>0’” into the beginning of the conditional statement, we will go through only the branch with “x>0”. It automatically creates dead codes for the other branch as they are unreachable，as shown in Figure 16. Currently, the result of the end of the procedure displays information only relevant to the if-branch.

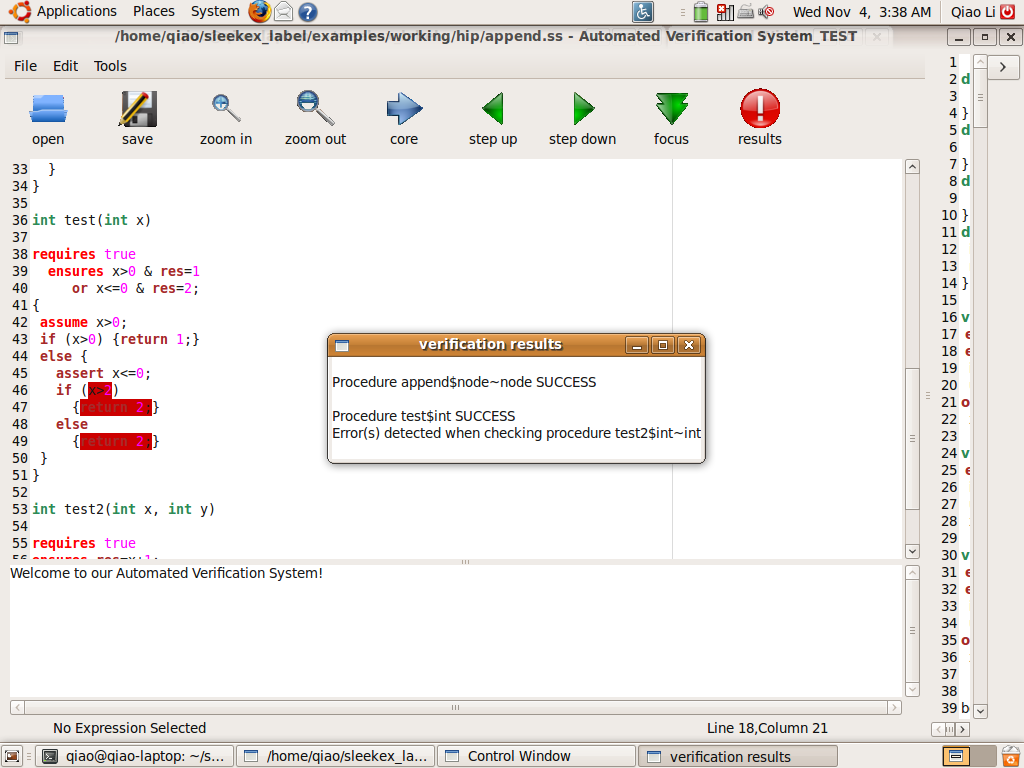


Figure 17. Command of assume sample

1. Program States

Procedure $append has two pairs of specifications. If we place the cursor in this procedure and press “step down” button, we can trace through the procedure. We have two sets of program states for each sub-expression due to two specifications. Take the example of the first sub-expression in Figure 17.

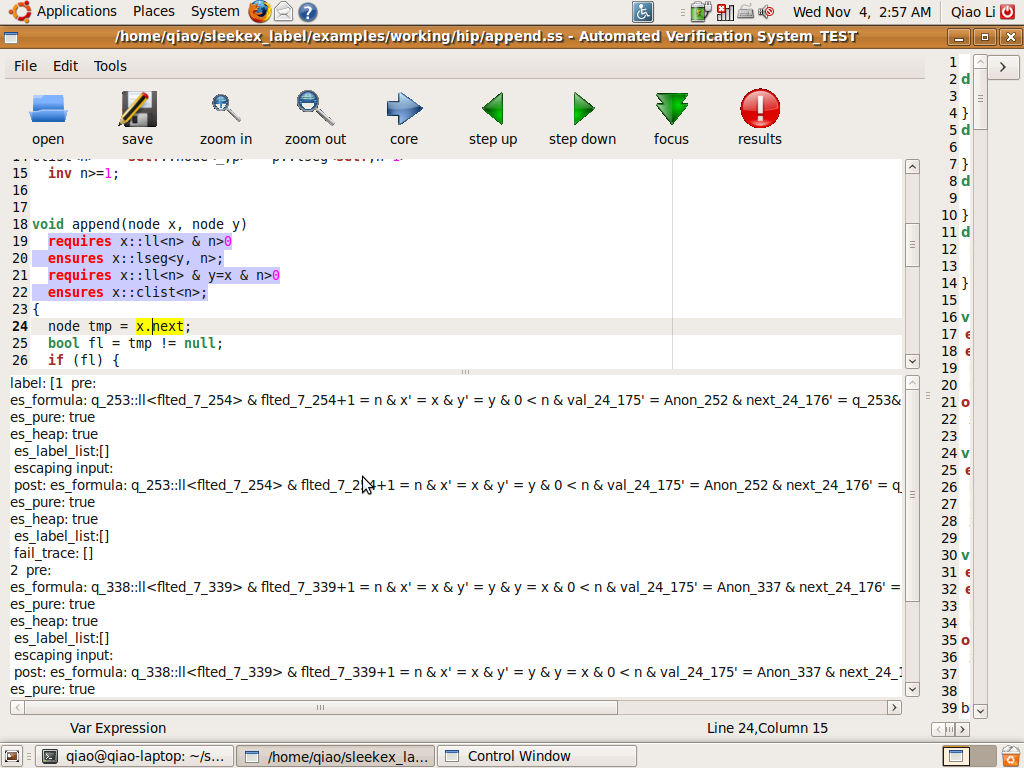
1. Focus on One Specification

Select the specification or place the cursor at that specification and press the “focus” button, as in Figure 18. Subsequently, the specification will be highlighted pink. The program states will only show the information relevant to that specification.

1. Core Window

The user can choose to minimize or maximize the Core Window to display the code representation of the source codes. The core program has been pretty printed for easier reading.

This selected sample is short and simple for demonstration purposes. It illustrates that we have used the information of the existing system to assist debugging through highlighting and displaying only relevant information. We have to admit that the function of program states is hard to interpret. However, it contains all the information of heap states at any program point. If the program is very long, going through codes line by line can be very tedious. Instead, using the information and highlighting, focus on only part of codes to interpret the program sates can be less tedious.



For second specification

For First specification

Failed traces

Heap states after execution of sub-expression

Heap states before execution of sub-expression

Figure 18. Sample information of heap states

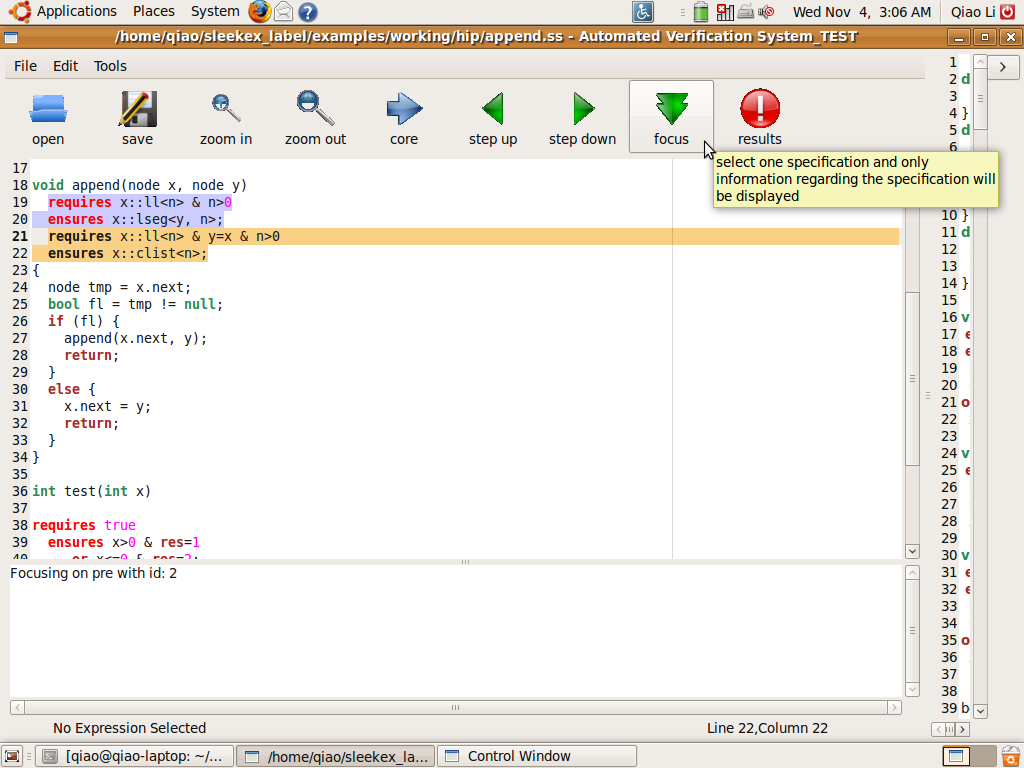


Figure 19. Select a specification to focus on

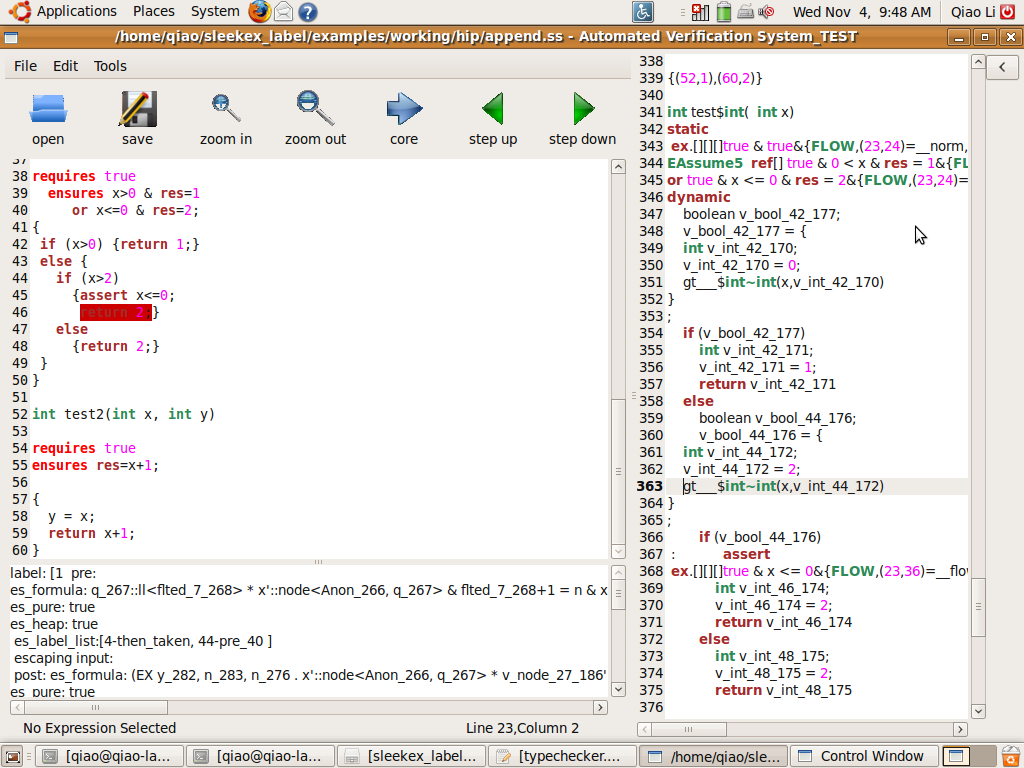


Figure 20. Core Window and core program