1. INTRODUCTION

Software is often too unpredictable to developers and users. Formal methods were suggested to be use to guarantee software reliability (Jones, O'Hearn and Woodcock, April 2006). At the very least, we hope to ensure that the software follows its specifications, as promised to its users. These formal methods, if well established, will be able to improve the reliability of both the existing and future systems by providing warnings of errors. Moreover, this verification with specifications can be used for static debugging, which has been a difficult goal for many years as well. Since we have a working verification system prototype with various debugging information, we are motivated to look into the possibility of a static debugger.

The existing system works on software verification through automated verification (Nguyen, David, Qin, and Chin, 2007). Based on separation logic, this verification system provides basic tools to automatically verify whether a given program code is able to correctly meet its annotations, including pre-conditions, post-conditions and invariants. Particularly, properties of the program, such as data structures, bag of values used can be specified and verified in the system. The prototype system has proven effective in many examples with various interesting data structures, such as linked lists, binary search tree and AVL trees, etc.

The system, currently, is able to verify whether a procedure follows its specifications and identify possible dead codes. However, the system is not yet able to provide programmers with the specific details and reasons of failures if the code cannot meet its specifications. If the program fails the verification, the user can insert a command at any program point to obtain the program states at that point for debugging purposes. The program state is represented by a function representing the heap status of all variables used in the verification. However, it is still a challenge for the programmers to interpret and understand each function during the debugging process. Also, to identify the possible reasons of the failures, the programmer needs to trace through the program and analyze the program states at all possible points of failures.

Moreover, there is no proper graphical interface to edit the code and display the

pieces of information. The users can only insert checking codes and execute the system several times using command line. This makes the programmer's job tedious and slow, and it can be even tougher when the program is complex and there are multiple specifications for each function.

Furthermore, if we are unable to display the useful debugging information in a clearer way, users may not be able to identify them. Therefore, we propose to explore how the verification information can be used for debugging and develop a graphical interface for our verification system to display program dynamics for potential static debugging. This can make better use of the information and functionalities in our verification system.

Nevertheless, automated debugging has been a very difficult goal to achieve at the states of arts (Ducasse, 1993). The current techniques can only approximate the location of the errors. Most existing debuggers can prove limited problems, such as detection of pointer errors in C programs (Gaugne, 1997). Our system can identify possible errors at compile time. Also, much information in the existing system is useful for debugging. With our automated verification system, the potential static debugger will have a greater completeness. Our system uses the "verification with respect to specifications" strategy under the classification of (Ducasse, 1993). The debugger is considered static as we focus on the source codes, not runtime execution traces. This provides a greater completeness because runtime execution is limited to a set of inputs, but not all possible inputs.

Currently, the prototype of the GUI contains basic operations on editing the source codes and visualizing the program dynamics. It displays the debugging information such as the static program status at any program point, helps users to trace through the source code and links to core representation of the source code in a more user-friendly way. Besides, the debugging interface developed in this project can help the further development of our automatic verification system because information is more accessible and organized.

In Section 2, we will give more details about the existing automated verification system and its basic architecture. Considerations due to our current needs and other

similar work will be present in Section 3. Section 4 contains the implementation and the main features of the prototype interface. Section 5 talks about limitations and future development and lastly, we will conclude in Section 6.

2. BACKGROUND OF THE EXISTING SYSTEM

The main focuses of this project are on making use of the information in the verification system for static debugging and visualizing them in a user-friendly way. We will show the basic procedures and features of the system through examples, but not going into the separation logic or the validity of the verifying process, which can be found in (Nguyen, et. al. 2007).

2.1 An Overview of Our Automated Verification System

2.1.1 User-definable Predicates

The prototype of the verification system has a few features. First of all, it can represent a wide range of data structures, such as AVL-tree and sorted list, using functions. The detailed properties of the data structures, such as size and bags of values allowed, can be specified. For example, we define the non-empty sorted lists as follows:

$$sortl\langle n,min,max\rangle \equiv (self::node\langle min,null\rangle \wedge min = max \wedge n=1) \vee (self::node\langle min,q\rangle * q::sortl\langle n-1,k,max\rangle \wedge min \leq k)\ \mathbf{inv}\ min \leq max \wedge n \geqslant 1.$$

As above, "*self*" denotes a root pointer to the specified data structure and is necessary for nodes to be accessible and well-founded. The function indicates either the list contains one node with values (*min* and *null*) or one root node followed by another sorted list. If it contains only one node, value min should be the same as max and size of the list is one. If there is more than one node, value of the root node should be smaller or equals to the minimum of the following sorted list. We also declare $min \leq max \wedge n \geqslant 1$ to indicate the invariant of the list. Any data structure that does not conform to this function inside a program cannot be recognized as a non-empty sorted

list. If the user indicates the procedure has a sorted linked list with certain size as a postcondition, the program status at the end of the procedure should contain a formula with such a sorted list.

2.1.2 Verification Process

Our system uses a set of forward verification rules based on separation logic. There are two parts that we mainly need to check. At each call site, at least one of the preconditions of the call methods has to be satisfied. Also, for each method declaration, we start with an assumption on the precondition, but expect that the declared postcondition will be satisfied on exit of method. For each procedure, we can have multiple preconditions and their corresponding disjoint postconditions. All the postconditions and invariants for any precondition should be verified.

Using the predicates, the programmers can indicate their specifications for each function to verify. If the function is working as they predicted, the system will be able to indicate success in verifying the function.

The forward verification is using Hoare-style triples. During the verification process, the program state at each program point is stored. Taking the example of the following function of inserting a node to a sorted list:

```
node insert (node x, node vn) where
    {if (vn.val ≤ x.val)
        then {vn.next :=x; vn}
     else if (x.next = null) then
            {x.next:=vn; vn.next:=null;x}
     else {x.next:=insert(x.next,nm);x}}
```

The precondition of this function is as follows:

$$\{x'::sortl<n,min,max> * vn':: node<v,\_>\}$$

It indicates a sorted list and a single node with value v. The program states for the first line of the function would be:

if (vn.val ≤ x.val){

$\{(x'::node<min,null> * nv'::node<v,\_> \wedge min = max \wedge n= 1 \wedge v \leq min) \vee (\exists$ $q,k.x'::node<min,q>*q::sortl<n-1,k,ma>*nv'::node<v,\_> \wedge min \leq k \wedge min \leq max \wedge n \geq \wedge v \leq min)\}$

We can see that the program state is based on the specifications and the user defined predicates. Subsequently, the state will be updated after each statement. The final states at the end of this function will be compared to the declared postconditions to determine correctness.

As mentioned in the introduction, this information can be helpful in debugging because the user can find out whether at this program point, the data structures are the same as expected. In our system, we can get this information by inserting a “dprint” command in the source code. And this command will print the heap state of the program at the correct program location.

2.2 The Architecture of Our Automated Verification System

When a source code file is passed in the system, Figure 1 shows the basic procedures it goes through to verify its correctness. Input Abstract Syntax Tree (AST) is built according to the input. Core AST adds in primitive predicates, such as Boolean and integers, and simplifies Input AST for type-checking and instantiating. The simplified AST contains fewer constructs such that it is easier to form verification rules. Also, more information about the program status is added into Core AST during verification by the verifier. Input AST contains the structure of the input, while Core AST includes the useful debugging information.

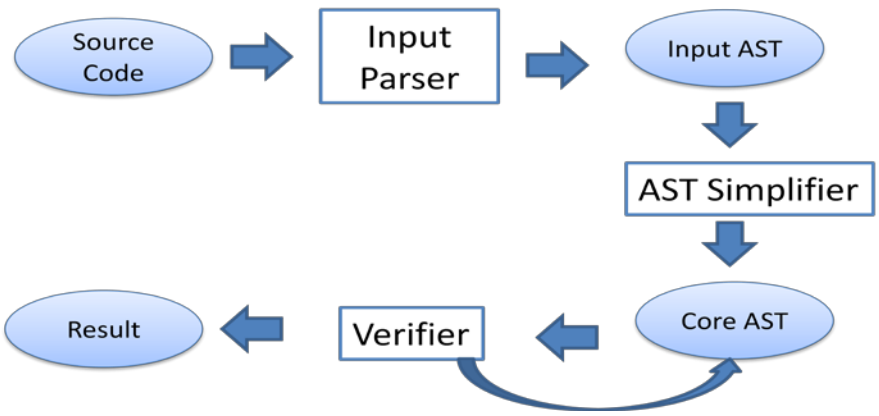

Figure 1. Verification System Architecture

The prototype system is built using Objective Caml. The verification of size and length can be done using automatic provers of Omega Calculator, Isabelle, and

MONA while Isabelle and MONA can verify bags or sets of values. Objective Caml has a strong static type system and it combines the advantages of functional programming languages and object oriented approach.

Currently, the system has been well tested with basic data structures such as Binary Tree, Linked lists and heap, as well as some basic operations on these data structures, including insertion, deletion and sorting. In the future, the system will work on more complicated and larger programs with mixed constraints.

3. DESIGN CONSIDERATIONS

The main aim of this project is to use the inferred information from the existing system to facilitate debugging. When we are developing the system, there are troubles that we faced due to the lack of graphical user interface (GUI). Therefore, based on our experience with using the verification system, we have a list of basic features that we wish to implement first for better use of the existing system.

Firstly, the users should be interacting with the system through the source codes. In program visualization, (Diehl, 2007) advocates the use of diagrammatic representations, such as control flow diagram. GamaSlicer, which uses contract-based slicing to facilitate verification, displays the AST of the program (Cruz, Henriques, Pinto, 2009). However, we feel that these representations of the whole program can be too informative and bulky. It is not very natural for the programmers to debug with these diagrams as eventually, they will still need to go back to the source code to understand and remedy the problems. Control flow diagram can be helpful to visually understand the logic and connections between procedures or large systems, but may not be useful for debugging of individual programs. In our opinion, it may still be better to focus on stepping down the source code or obtaining information by clicking on the source code. This is familiar and intuitive.

MrSpidey is a static debugger for DrScheme, Rice's program development environment for Scheme (Flanagan, 1997). Though MrSpidey uses componential set-based analysis to identify possible unsafe operations, it has a similar intention of providing the programmer useful information from its analysis in a nature and easily

accessible manner. Hence, it uses program mark-ups to present the information. This means there is no major change in the structure of the program, but mark-ups, such as font and color changes, are used to provide necessary information from its analysis. We would like to do some similar program mark-ups as well.

While building the GUI, we would like to retrieve the information from our system but make minimum changes to it. As the existing system is developed using Objective Caml(OCaml), in order to well integrate into the system and better cater to our current needs, we chose to develop the interface using the graphical user interface(GUI) libraries in OCaml. The GUI library we chose is lablgtk, which is an interface to gtk+. It supports almost all, except for one, widgets in gtk+, including GWindow, GPack, GButton and GMenu, ect. One of the most important widgets is the SourceView widget. This widget helps to display source codes with syntax coloring and line numbers. Some text editors, such as gedit in Ubuntu, have used gtkSourceView to facilitate their display of text. In the following section, I will elaborate on the main features of the current GUI.

4. IMPLEMENTATION

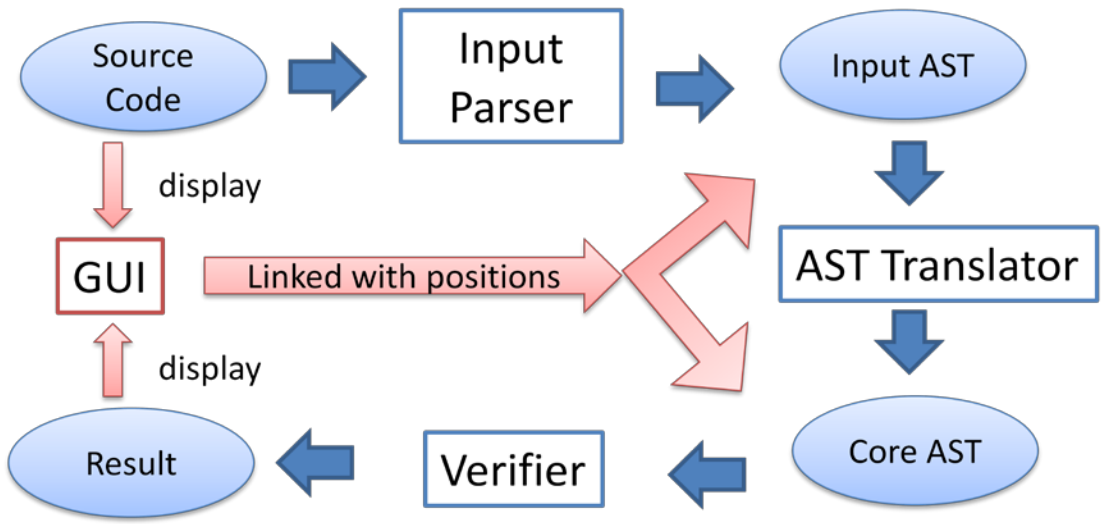

Figure 2. Integrating GUI to the existing system

4.1. Correlating Input AST and Core AST

The users will mainly work on the source codes in the debugger to find out

reasons of failures. They will need the information of each program point in the source. In the system, we can understand the structure of the source codes using Input AST as the source codes have been parsed to build Input AST. However, the debugging information is stored in Core AST during verification. Therefore, there is a need to link the Input AST to Core AST. This means we will be able to know which nodes in the Core AST the user is referring to when he or she clicks on the source codes. Subsequently, we can retrieve the appropriate stored information in Core AST for debugging.

To tackle this problem, we have added the location of the source codes as another field in every expression of Input AST. In the field, there are three variables, namely start_position, middle_position and end_position. (The previous system uses only the location). This is done during the parsing of the source codes without many changes in the existing codes. For example, for an assignment expression:

```
k = k+1;
```

The start_position is just before k, end_position is just after 1 and middle_position is at sign "=". Also, these positions are stored in Core AST when Input AST goes through AST Simplifier. Therefore, when we need to find the corresponding nodes in both ASTs or correlate Core AST and Input AST, the position of the nodes in the source file will be the essential link between core program and source program.

4.2. Use of Labels

Originally, Core AST contains the program states at each program point. However, one program points can have multiple program states. The functions can be messy, due to two situations in the code: 1. Multiple specifications; 2. Branch points.

4.2.1 Multiple Specifications

The users can specify more than one preconditions and their corresponding postconditions can have multiple possibilities as well. As our static program state is based on one precondition, as mentioned in Section 2.1.2, one program point can have multiple program states when there are multiple preconditions. In this case, it is possible that only one of the pairs of precondition and postcondition causes failure

where the rest can be verified. In order to provide more information to the user regarding each individual specification, we need an identifier for each specification in the ASTs and this identifier is added into the program status in Core AST as a label. Hence, we are able to know the specification that one set of program states belongs, and the exact location of this specification in the source file. Subsequently, if only one of the specifications causes the failure, the user may wish to focus on only that specification and only program states pertaining to the specification will be displayed. The details of this feature will be discussed in the Section 4.4.

4.2.2. Branch Points

Similarly, branch points can cause multiple program states. In our case, branch points refer to three cases: the if-statement, the try-catch statement or the method calls. One of the branches can be the only one causing the failures, but not the rest. It is useful to know the program state or failure is caused by which branch. The user can then concentrate on investigating the respective branches or method calls.

Each branch point is tagged with a unique identifier and for each program state; it contains all the previous branch points, which have been taken. The users can then trace back the branches that may cause failures.

Besides, labels provide extra information for the GUI to indicate to the user regarding the specific specifications or paths that have caused failures. For example, highlighting the problematic specification with a different color is one of the features in the current debugger. This will be shown in Section 4.4.

4.3. Program States (Using “dprint”)

Each point at the source code belongs to one of the sub-expressions, represented as nodes in ASTs. Referring to figure 2, sub-expression is from position p1 to position p10. When we are enquiring the program states at any of the position in between this sub-expression, they should all refer to the program states of this sub-expression. For each specification, there is a different set of information, including a state function just before the execution of the sub-expression, a state function just after the

execution, a label list with taken branch points and a failed trace if failure occurs. The state function is the kind of functions that we have mentioned in Section 2.1.2, while label lists include the branch points taken with their identifiers. For a failed trace, it is only available if there is a failure occurs at this sub-expression (indicated as a new fail), or a failure occurs at an earlier point in the procedure (indicated as an old fail).

A detailed example of the program states can be found in Appendix-A, Section 7.

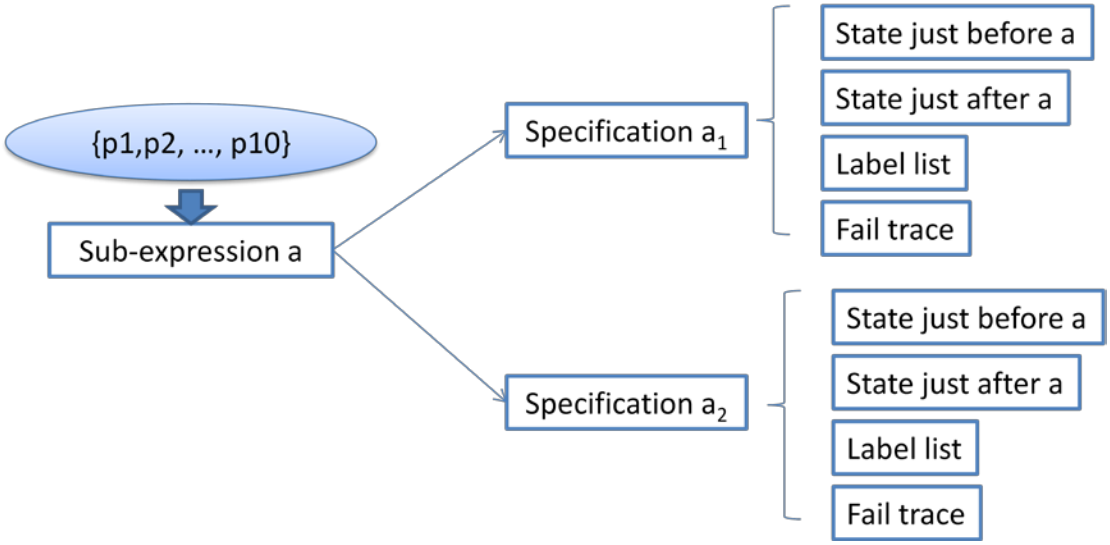

Figure 3. Illustration of program states for a sub-expression

4.4. Use of "assume" and "assert"

Another way to help debugging is adding in "assume" commands and "assert" command in the source codes, where appropriate.

The command of "assume" can indicate an assumption of the current data structures and values of any variables. For example, a procedure may have a precondition of "$x \geqslant 0$". All the program states are created and updated according to this precondition. If the user finds that the main problem lies in the boundary of $0 \leq x \leq 1$, he or she may wish to add in "assume $0 \leq x \leq 1$ " to look into only this range.

This can be very useful in the cases of branch points. As mentioned in Section 4.3., branch points may cause multiple program states at the end whereas only one particular branch causes the failure. In this case, the users may add in the command

“assume cvar” (where ‘cvar’ is the Boolean function of that branch) to that branch so that ‘cvar’ is always true and the system will only investigate and verify according to the intended branch.

On the other hand, “assert” command check some properties in the middle of the source codes. With the “assert” command, the system will check whether the problematic states fulfill the stated property. If the stated property is wrong, most likely some errors have occurred before the point of this command.

4.5. Main Features of the Static Debugger

We will highlight some motivations and desirable features for our proposed system to assist the static debugging process in this Section.

Wish Item 1: Provide a basic platform for displaying source codes and running verification system together

One of the main troubles during debugging is that the user has to go back and forth between the source code and the system. When the verification fails, the user has to edit the source code, add in necessary “dprint” commands for program states and then rerun the system through text commands. It might be more convenient to integrate the basic editing of the source code into our system for easy debugging.

Feature 1: An editor with basic editing functions and rerun the verification system automatically when a file is opened or saved.

There are three main windows in the debugger, as shown in Figure 3. The source window displays the source code; the core window displays core representations; the information window displays useful debugging information.

Using the GSourceView in lablgtk library, we have defined the customized syntax highlighting for our source files. Line numbers of the source are also shown. The users can directly create, open, save a source file, or find a string within the file.

Also, as the system is run when a file is opened, the core program of the source is generated and displayed. Our core program means the translation of readable codes from Core AST. Core AST contains the constructs that go through verification rules. If the users wish to look into how the input has been translated into Core AST and

verified, they can refer to the Core Window. There is pretty printing of the core representation for more pleasant viewing. However, we feel that users may not be very interested in the Core Window, because it is more natural to debug in source. Therefore, the Core Window is closed when the GUI is first opened and user can open it with the button on the top right corner. For the purpose of further development of the system, the developers of our system may wish to refer to the core program to check on Core AST and verification processes. Presenting both source and core representations in the same GUI can ease their jobs.

Moreover, when a file is opened or saved, the users do not need to request the system to analyze the source again. Instead, the opened or edited source code will be automatically verified by the system. The verification results will be displayed in a pop-up window, as shown in Figure 4. The results show whether each procedure has been successfully verified. The users can refer back to the verification results by clicking on the “results” button. If there are dead codes, which mean that some codes are unreachable, they will be highlighted in red immediately after verification.

Wish Item 2: A more direct way to show program states rather than inserting “dprint” command

This motivation has been mentioned in the above sections. The program states are very useful and handy for debugging. However, inserting into the source code is very troublesome and if we wish to get all the program states, we will have to add in “dprint” for every expression.

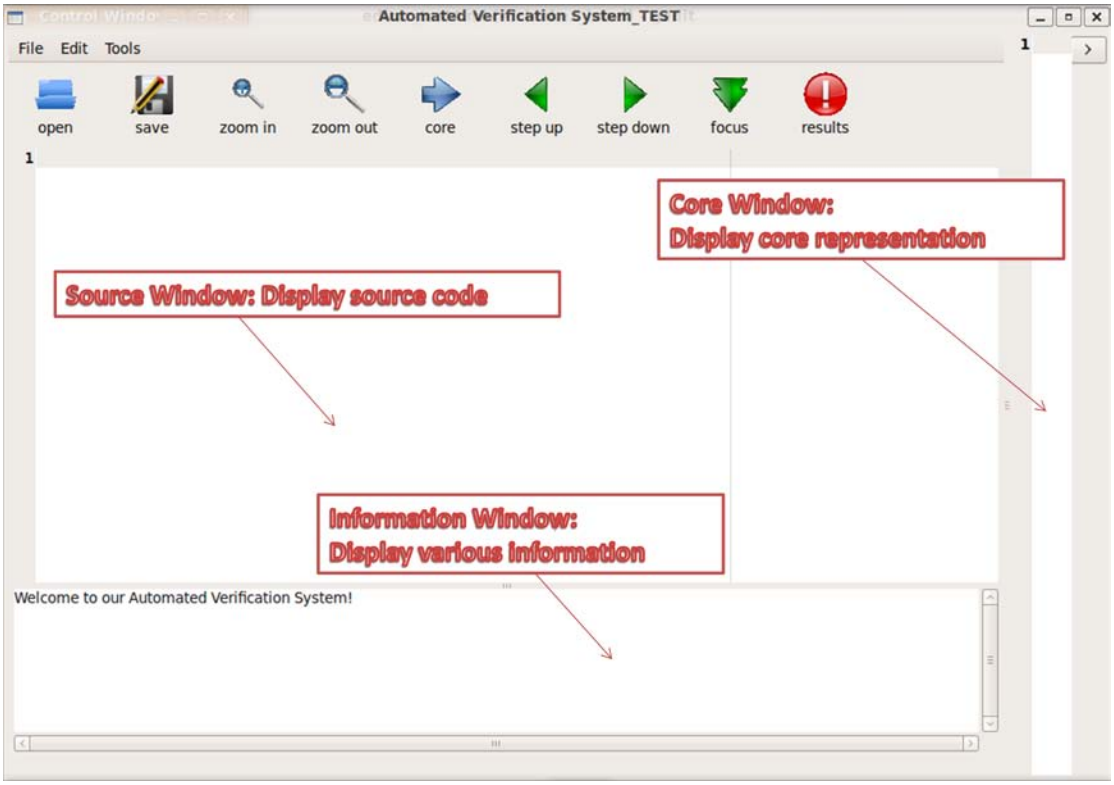

Figure 4. Three main windows of the GUI

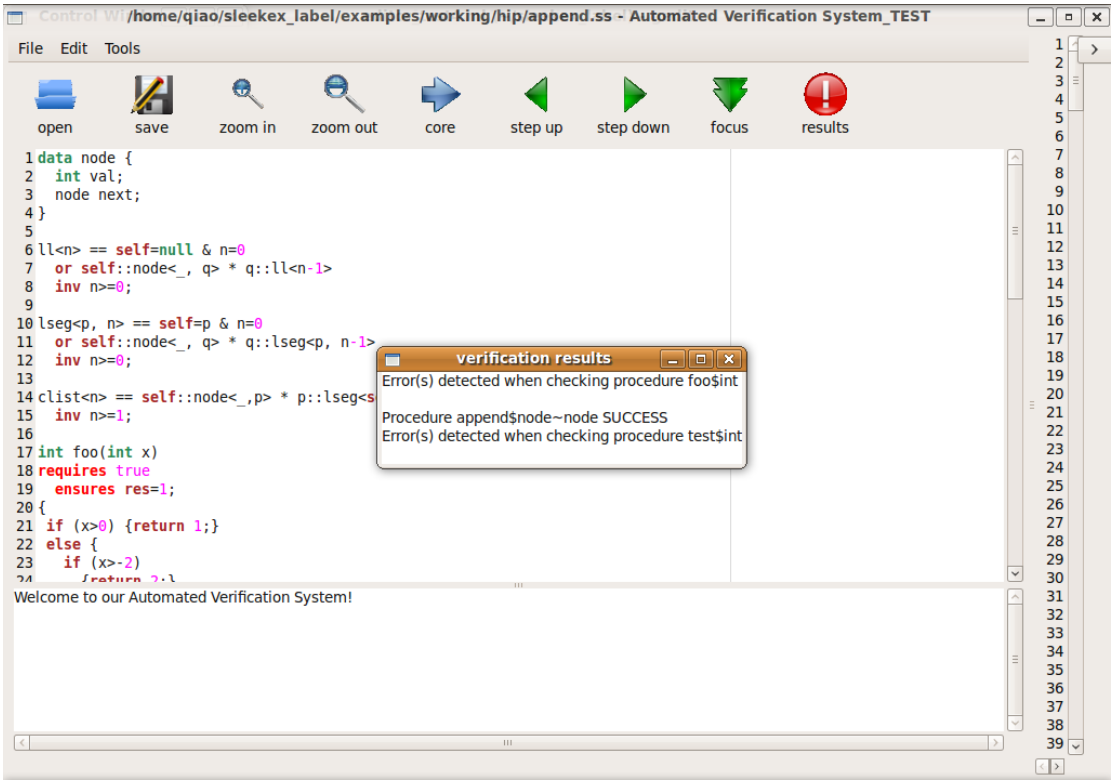

Figure 5. Verification Results Window

Feature 2: Displaying the program states of any position in the source codes and highlighting the corresponding sub-expression with different colors

When users click on any point of the source code or move the cursor to that point, the debugger will retrieve corresponding program states of the point from Core AST and display in the Information Window. At the same time, this sub-expression is highlighted in yellow to indicate that the program states belong to it, if there is no detected error at this point. However, according to the fail trace, if there is a possible failure caused at the execution of the sub-expression, the expression will be highlighted in red color instead. If the failure is caused by codes before the expression, we have pink highlighting of the expression. It should be clear to the users that they need to investigate more into those expressions with warnings of red or pink colors，as in Figure 5 and Figure 6.

Here, the corresponding program states of the position refer to the states of the smallest sub-expression in Core AST that includes this position. The type of this sub-expression is at the left bottom corner. This is because one position may be inside more than one sub-expression. Again, take the example:

```
k = k+1;
```

If we refer to the positions just before and just after the highlighted "*1*", these positions belong to three sub-expressions: one for the sub-expression of an int constant *1*(contains only '1') ; one for that of the assignment of variable *k* ("k= k+1"); it is also part of the whole procedure. We chose to display the smallest sub-expression, in this case, only the constant 1, because we have to ensure the smallest sub-expression has an expected program state before going to the larger ones.

The needs for investigating all the sub-expressions sharing the same position lead to the third feature.

Wish Item 3: Ability to obtain the program states of all sub-expressions contain the same position in source

Feature 3: Zoom in/out on the different sub-expressions, which contain the same position when the cursor is stationary.

Based on Input AST, without moving the cursor, we can zoom out to get the

information of larger sub-expressions inclusive of the cursor position, and zoom in to obtain the smaller sub-expressions, as shown by Figure 6. The text in the Information Window displays the program states of the current highlighted expression.

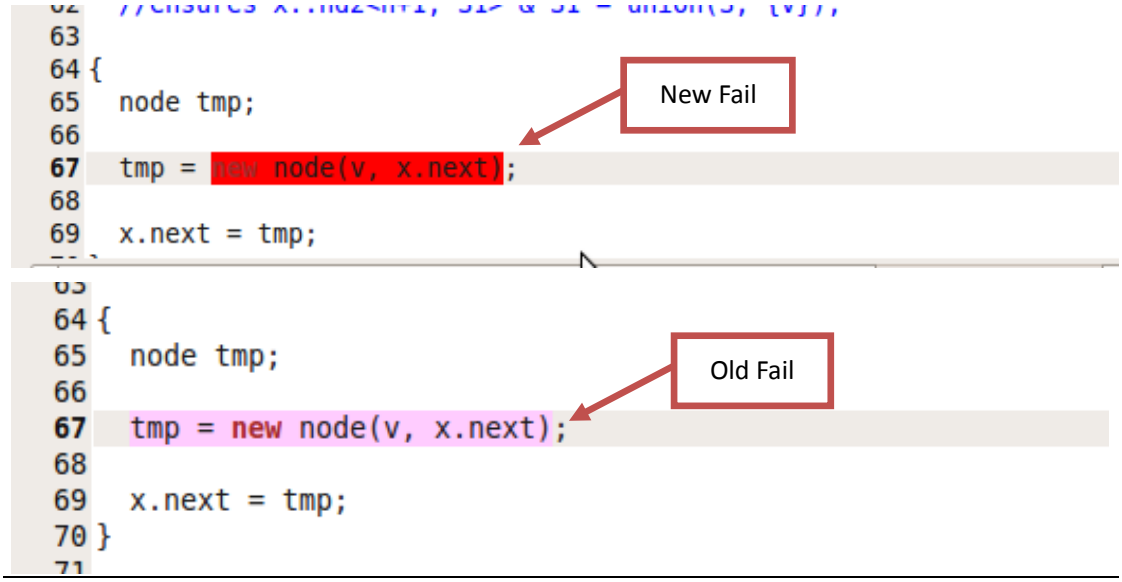

Figure 6. Highlighting expressions with different colors

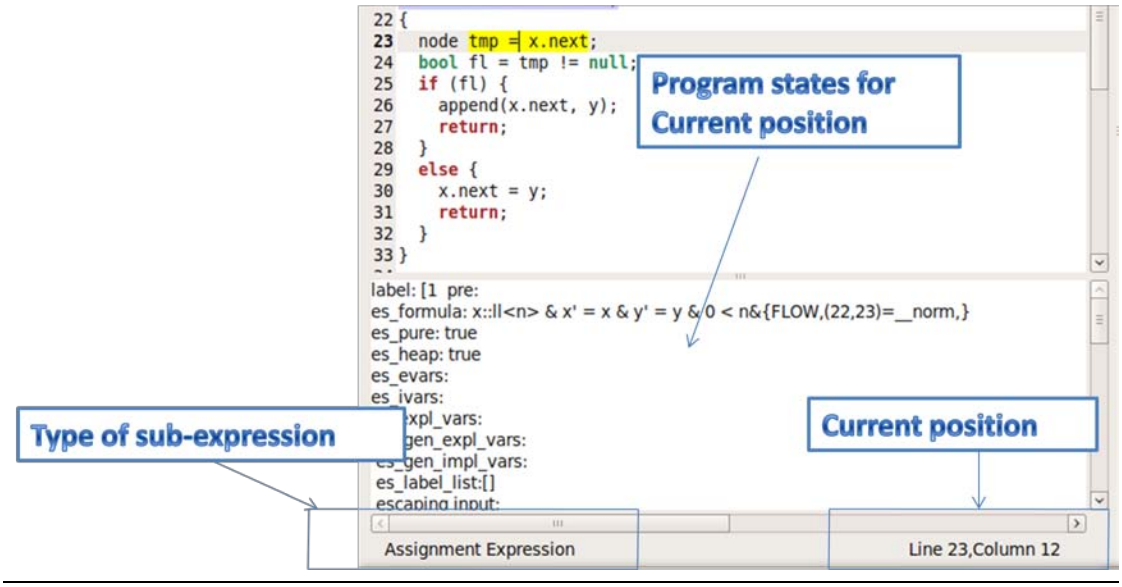

Figure 7. Displaying program states and highlighting corresponding sub-expression

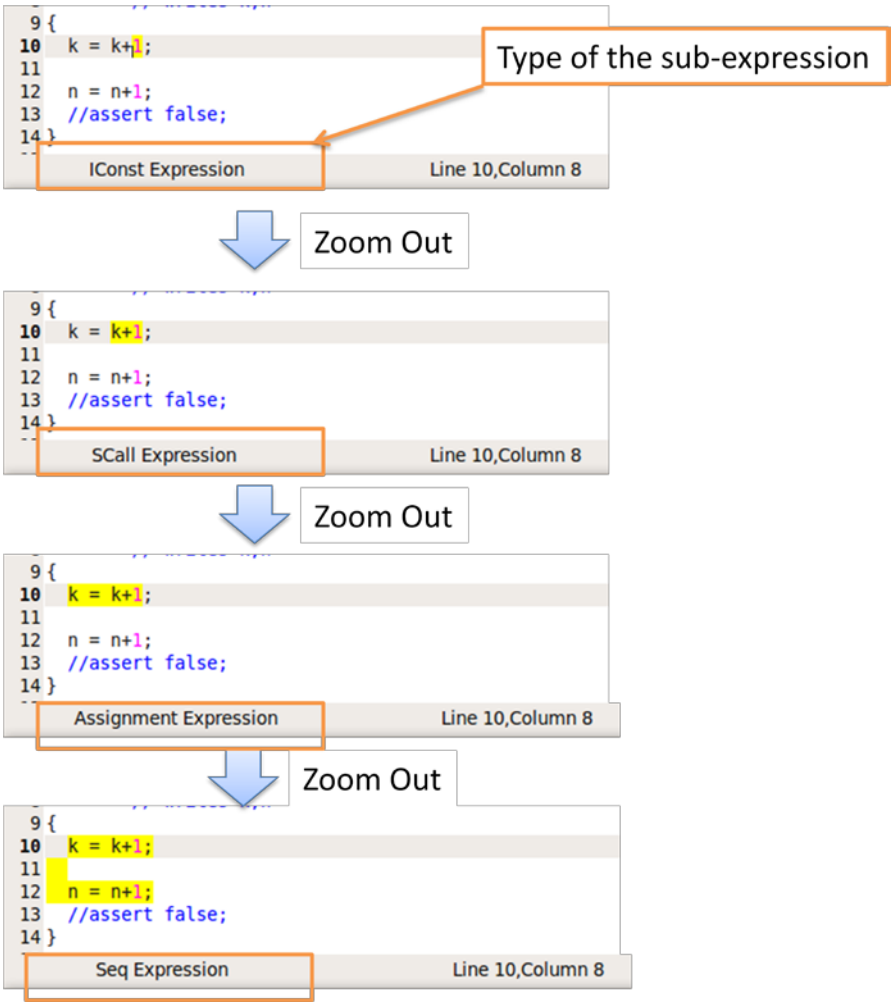

Figure 8. Zoom in/out at the same cursor position

```
52  //********************** MULTI PRE/POST **************************
53
54  requires x::hd<n> & n > 0
55  ensures x::hd<n+1>;
56
57  requires x::hd1<S> & S != {}
58  ensures x::hd1<S1> & S1 = union(S, {v});
59
60  //********************** SINGLE PRE/POST **************************
61    //requires x::hd2<n, S> & S != {}
62  //ensures x::hd2<n+1, S1> & S1 = union(S, {v});
63
64 {
65  node tmp;
66
67  tmp = new node(v, x.next);
68
69  x.next = tmp;
```

Figure 9. Highlighting the problematic specifications with different colors

Wish Item 4: Indicate the specifications that cannot be successfully verified

As mentioned in Section 4.2.1, if we have multiple specifications, we wish to know those that cause verification failures and this information helps us further study the reasons of failures.

Feature 4: Highlighting the specifications relevant to the current sub-expression. Erroneous specifications are highlighted in pink while the rest is in light blue.

Using the label information store in the sub-expression, we look for the corresponding specifications with the labeled identifiers. The fail trace list indicates the identifier of the specifications that cause the failure and it is highlighted with pink color.

Referring to figure 8, the second specification fails at the red-highlighted sub-expression.

Wish Item 5: Focusing on only the erroneous specifications

When we realize that not all the specifications cause errors, we can in fact focus on the program states from those erroneous ones, because having too much information in the Information Window can be unclear.

Feature 5: Selecting one of the specifications and only information relevant to this specification is displayed

The users can select the desired specification by highlighting part or the whole of the specification and click on the “Focus” button. Subsequently, when the users click on any position in the program, only program states on that specification are displayed. The information is less lengthy. More details are given in Appedix-A.

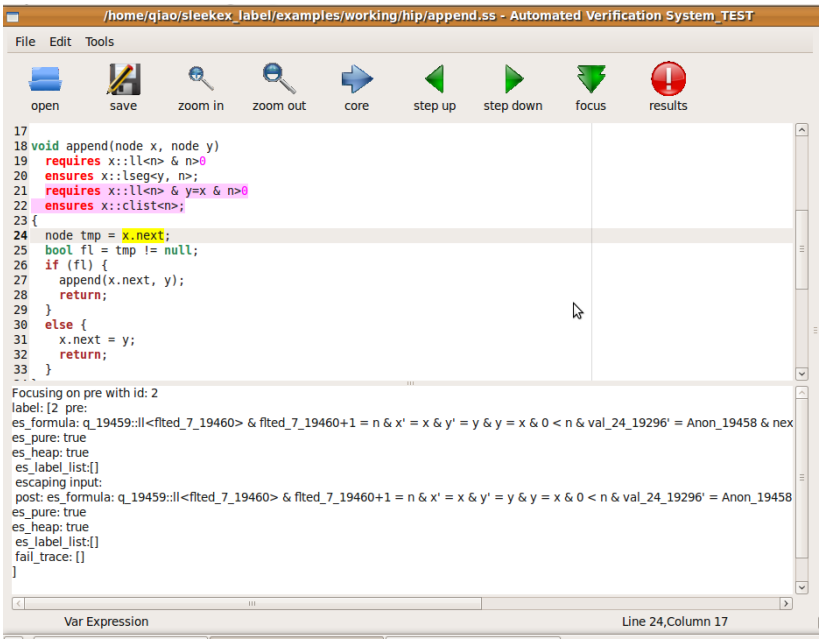

Figure 10. Focus on only one specific pre/postconditions

Wish Item 6: Help the users to trace the source codes

Though the users can trace through the codes by moving the cursors, moving the cursor can be slow when the expression is large and it may be inaccurate if they move the cursor too fast and miss some sub-expression. The users, therefore, may wish to move down the sub-expressions. In a more systematic way, we want to provide commands to step down/up the source codes.

Feature 6: Step down the source codes by expressions

The users need to place the cursor in the desired procedure they want to investigate and press the “step down” button will start the stepping from the beginning of the procedure. They can also move back to the previous expression by pressing “step up” button. The sub-expressions and specifications are highlighted accordingly and the appropriate programs are displayed.

Wish Item 7: Able to indicate the corresponding core codes of the sub-expressions

Core representations are the code representations of Core AST.

A more direct link between source and core representations visually can help users to understand core representations and the verification processes. Especially,

these can be very for our own developers of the system because they can find out whether they make any mistakes in the AST simplifier or how they can improve on the verification process.

Feature 7: Relating the procedures between source and core representations

Currently, this is a very limited solution for Wish Item 7. By clicking on the "core" button, the portion of the core program corresponding to the procedure where the cursor lies will be highlighted in blue. When the Core AST is translated into string, we have included additional marks in the pretty printer to identify the positions of the procedures in the buffer. In order to identify all the sub-expressions, we need to add in more marks for each sub-expression with unique identifiers.

However, most users may not like to refer to the core representations when they are unfamiliar with the core. Hence, we have put this as one of the features for future developments.

5. Limitations and Future Developments

The static debugger is a first step and we will need to continuously add in new features and information to enable better debugging experience.

One of the important features we wish to include is that when we are stepping down the source codes, we can always give users the choice of a tracing through only a particular branch at branch points. This means we can automatically add in "assume" commands at branch points without changing the source codes. Furthermore, users should be able to go back to the branch points to make a different choice and continue with a different branch when necessary. At other program points of the source, users may also wish to add in "assume" or "assert" commands without physically altering the source. This is useful because users do not need to go back to the source codes after debugging to remove all the "assume" or "assert" statements.

Despite of the clearer indications of failure paths through highlighting and labels, the functions of program states remain relatively difficult to interpret due to the long and random variable names. Please refer to Appendix-A for an example of the functions. Therefore, we will work on simplifying these functions, or display them in

a more vivid way, such as using an animation to represent different data structures.

Other features, such as improving on relating the source codes with the core representations are necessary to be considered as well. Besides, we can use slicing to improve efficiency and effectively with both code analysis and verification.

Currently, the GUI is built mainly for our internal usage and catering to our own needs. For convenient integration, libraries of OCaml have been used. It is also possible to have an extension for our system in other Integrated Development Environment (IDE), for instance, the Eclipse. We can have more comprehensive functionalities for editing and managing large pieces of source codes in IDE. The current GUI provides us with experiences about necessary features for further development in other platforms.

Also, we have assumed the correctness of the syntax of the programs and specifications. The current parser will inform the users of the error location but not the reasons of errors. To be a more effective debugger, we need to incorporate a stronger compiler to identify syntax errors.

Another limitation that we have to note is the scope of debugging we can do. Though our verification system has proved some basic algorithms, such as sorting, effectively, many other algorithms or specifications not in our examples (Nguyen, 2007) may not be effective. A false negative on verification is possible. However, as most static debuggers cannot provide completely accurate information, the users still need to decide on how to correctly debug their programs.

Nevertheless, it is possible to use this static debugger as a tutoring system for specific programming teaching purposes. The students can test their understanding of data structures by learning to write accurate specifications and debugging their relatively simple programs.

With the basic interface, we will continue to refine on the completeness and accuracy of the verification system, while continue improving on the visualization.

6. Conclusion

In this project, we proposed and develop a GUI for static debugging for an

existing automated verification system. The main purpose is to display and present the information from the existing system in a more user-friendly way. Subsequently, the system is more convenient to use for verification and debugging. The existing features have been mainly based on our own experiences of using and developing the system. Basic features, such as editing, syntax highlighting, error indications and stepping down the codes, have been developed to cater to our needs. With the current support of basic data structures and operations, it is possible to assist in debugging of simple programs. Other features and integrating into an existing IDE to form a complete platform can be considered in the future.