Interactive Symbolic Visualization of Semi-automatic Theorem Proving

Chandrajit Bajaj, Shashank Khandelwal, J Moore, Vinay Siddavanahalli*
Center for Computational Visualization
Dept. of Computer Sciences and ICES
University of Texas at Austin
http://www.ices.utexas.edu/ccv/

Abstract

We explore a new symbolic visualization model for semi-automatic theorem provers. Mechanized formal methods are finding increased use in the design and development of complex hardware and software systems. Most proofs are presented in a textual format, with intermediate formulas possibly consisting of megabytes of data, which are difficult to analyze and understand. This paper introduces a preliminary visualization environment for semi-automatic theorem provers in an attempt to help users steer the theorem proving process. The environment provides synchronized multi-resolution textual and graphical views and direct navigation of large expressions or proof trees from either of the twin interfaces. We identify three levels of the proof process at which synchronized multi-resolution graphical and textual visualization enhance user understanding.

CR Categories: H.4, F.3 [Information Systems Applications]: Logics and Meanings of Programs—; D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures

Keywords: Symbolic visualization, Approximate tree matching, HCI

1 Introduction

Formal methods allow users to mathematically describe a system, and then verify that the system behaves according to a given set of specifications. In software engineering, artifacts such as specifications, design, and source code can be formally defined and verified, helping in the development of "bug free" systems. Software and hardware systems have become complex enough for the formal verification process to require the use of semi-automatic theorem provers.

Although the applications have become more complex, user interaction with theorem provers remains mostly text based. When a proof attempt fails, the user needs to diagnose the problem and then come up with new theorems, lemmas and hints to proceed with the proof. This requires a thorough understanding of the proof attempt. Theorem provers typically generate large amounts of text during proof attempts – making an attempt difficult to understand.

A command line interface is used with most theorem provers. Pretty printing and text based primitives like searching are the main tools available to help reduce or manage visual complexity. Since theorem proving is, by definition, concerned with the syntactic manipulation of formulas in a formally defined syntax, the focus on text is inescapable. But that is not an argument that all interaction needs to be text based. The challenge is to integrate text-based tools with visualization tools to speed comprehension and interaction.

The design of user friendly interfaces to such large and complex text based systems is a well known problem. We have made a preliminary effort towards identifying three levels at which we can augment the command line interface with visualizations:

1. The overall proof process can be visualized by displaying a graph of the theorems and sets of previously proved theorems (books) used during a particular proof attempt.

2. The structure of the proof can be visualized, by displaying the subgoals created at each step, and indicating which subgoals could be proved or not. Navigating and browsing through the structure of a proof tree may help the user better understand a proof attempt.

3. The visualization of a single subgoal is something that is much required. Examining failed subgoals is critical towards understanding why a proof attempt failed. Within the proof of critical subgoals, the user needs to understand the steps taken by the theorem prover. Following the progress of similar subgoals through a proof attempt is useful in this situation. Graphical visualization would help quickly identify similar subgoals and their locations within the overall proof.

The visualization at all three levels involves finding a spatial mapping for the data at each level - data that is inherently not spatial. From our knowledge of interaction with theorem provers, we believe we have developed a mapping that we think will be cognitively useful. Our visualizations should help increase the efficiency of current users as well as provide a friendly environment to help new users learn the theorem proving system.

In this paper, we introduce a hierarchy of levels at which we can perform synchronizd multi-view textual and graphical visualizations. Domain specific pattern matching is used to enhance user understanding of proof attempts. To ensure interactivity, we use fast dynamic synchronized reconstruction of trees.

In section 2 we discuss our test case theorem prover, the current interaction methods and quantify the output generated during complex proof attempts. Section 3 presents related work on visualization in theorem provers, tree visualization and tree matching. In section 4 we provide details on the visualization of theorems and books, proof tree visualization, and function visualization. Section 5 describes the synchronized multi-view aspect of our visualization environment. Section 6 describes case studies. The final section presents our conclusions.

2 ACL2 and its Current Visualization

We choose ACL2 [Kaufmann and Moore 1997], [Kaufmann and Moore 1994], an industrial strength theorem prover, for our case study.

ACL2 is a tool that can be used to model hardware and software systems and then prove properties pertaining to those models. It is also a general purpose semi-automatic theorem prover, with a simple dialect of the LISP programming language. Other examples of

*{bajaj, shrew, moore, skvinay}@cs.utexas.edu

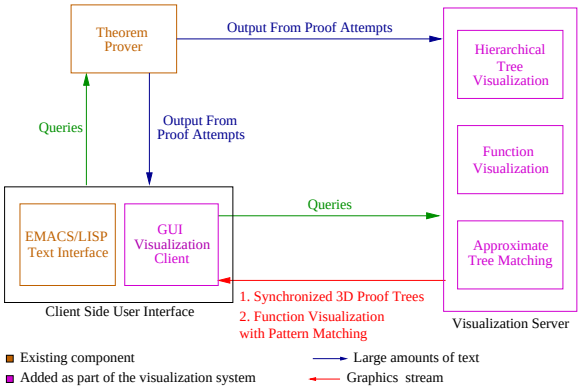

Figure 1: An Architectural Overview of our Symbolic Visualization System for Theorem Provers

similar theorem provers are NQTHM [Boyer and Moore 1988] (the predecessor of ACL2), PVS [Owre et al. 1992], and HOL [Gordon and Melham 1993].

2.1 Using ACL2

Symbolic manipulation is the method employed by ACL2 to prove theorems. It uses the axioms present in the logic and previously proved theorems/lemmas to prove the current goal. If a proof attempt fails, the user needs to understand the logical reason behind the failure, and then provide the system with hints, missing theorems or lemmas. This process requires a thorough understanding of ACL2 and the specific system being analyzed.

The proof of correctness of the kernel of the $AMD5_K86$'s floating point algorithm involved 1600 definitions and lemmas. ACL2 can produce terms 300,000 (20 Mb) lines long (as were produced while proving an equivalence condition related to the Motorola CAP digital signal processing co-processor). [B. Brock et al. 1996] discusses using ACL2 to prove theorems about these commercial microprocessors. The large amounts of text generated in such proofs is too cumbersome to comprehend even with powerful text manipulation and pretty printing tools. Graphical visualization makes comprehension of complex proofs easier by using higher spatial dimensions, a cleaner level-of-detail implementation and abstract representations.

2.2 Current ACL2 interaction

Most users run ACL2 in a shell running in a buffer inside an Emacs editor. The current user interaction is through a command line interface, with a read-evaluate-print loop. Emacs provides excellent support for searching, text manipulation and matching parentheses and other textual patterns. There is an option to obtain the proof tree and browse it in the same text model, allowing users to make simple queries on different subgoals. Although this system is powerful, it is entirely text based and relies largely on the discipline and Emacs skill of the user to impose structure.

3 Related work

We provide references to relevant previous work done in visualization of output from theorem provers. Tree visualization is required for the levels two (section 4.2) and three (section 4.3) and we present past work of interest. Tree matching is required in level three.

3.1 Visualization in theorem provers

[Thiry et al. 1992] discuss the need and requirements for a friendlier user interface to theorem provers, but do not attempt to visualize the information inherent in the proof process as a way to understand the proof attempt.

In [Goguen 1999], we see an attempt to use visualization to understand the structure of proofs and a complete system for developing a user interface. There are a few major differences between their system and ours. Their system is designed for users that read proofs (as opposed to specifiers or provers). The interface is through web pages that explain the proof with links to background material and tutorials. Their system is also designed with distributed co-operative collaboration between users in mind. Our system is designed to be used by theorem *provers*, working alone.

3.2 Tree visualization

There has been a significant amount of research in the visualization of data, and specifically in tree visualization. In [Keller and Keller 1994], there are a number of generic techniques and hints to visualize a variety of generic data (including time varying, animated, volumetric models). [Tufte 1990] describes a variety of commonly used data visualization techniques, presenting methods to increase our understanding of data by better representation. [Brown 1987] describes a system for algorithm animation, where the user can observe an algorithm's behavior through time. Algorithm comparison methods are also presented.

Complexity in tree visualization is reduced using hierarchies or culling of irrelevant information. Cone trees [Robertson et al. 1991], [Carriere and Kazman 1995] are a popular 3D visualization technique for trees. Space filling curves and fractals are other tree representations, but can lose structural information. [Shneiderman 1992]'s space filling approach is an example of maximum utilization of screen space but tree maps do not provide structural information with the same ease as a more traditional node-link approach. [Koike and Yoshihara 1993] improve on cone trees by using fractals to represent self similar nodes in a hierarchy. They give examples of browsing a large Unix directory structure. [Robertson et al. 1991] define traditional cone trees and the use of animation to reduce the cognitive load on users when they select a new node to focus on. [Carriere and Kazman 1995] try to improve the clutter present in cone trees as the size of data grows large by adding visual cues, hierarchical models and filtering to reduce size of rendered trees. In another attempt at the same goal, [Jeong and Pang 1998] describe Reconfigurable Disc Trees which alleviate some of the problems with traditional cone trees by using a disc as the basic shape instead of a cone. A new technique for rendering trees

was introduced in [Lamping and Rao 1994]. They use hyperbolic spaces to render arbitrary sized data in a limited space. [Munzner 1998] shows how large graphs can be represented and rendered in 3D hyperbolic space by laying out spanning trees.

Previous works on 2D tree layout algorithms, from which we derived our layout algorithm, are: [Walker 1990], [Reingold and Tilford 1981] and [Frank Van Ham 2001].

In section 4.3 we explain the need for tree matching in supporting expression visualization. There have been a lot of recent improvements in tree matching algorithms. In [Kosaraju 1989], the idea of convolutions between trees and strings, suffix trees of trees and don't care symbols are used to obtain an algorithm which has a cost of $O(nm^{0.75}polylog(m))$ where m is the size of the pattern tree and n is the size of the text tree. *polylog* is defined as the a polynomial of logs. Using substrings appearing periodically in the pattern tree [Dubiner et al. 1994] improve the $O(nm^{0.75}polylog(m))$ bound to $O(n\sqrt{m}\ polylog(m))$. In a series of papers, Cole et al. [Cole and Hariharan 1997], [Cole et al. 1999] show bounds of $O(n\frac{log^3 m}{loglogm}+m)$, a randomized algorithm of cost $O(nlog^3 m)$ and then later claim a deterministic algorithm at the cost of $O(nlog^3 n)$. In [Luccio et al. 2001], we see one of the first polynomial time algorithms for unordered exact matching.

4 Symbolic Visualization

In this section we provide details and justifications for the visualizations (and the types of visualizations) at each level of the hierarchy.

Figure 1 presents an architectural overview of our system. Taking an idea from [Thiry et al. 1992] our visualizations are generated by a separate server which communicates with the theorem prover through a simple protocol. The protocol can be implemented in the theorem prover's language providing access to information, as if the visualization server is a regular user.

4.1 Visualization of theorems used

A theorem is proved by using previously verified theorems and lemmas. Sets of theorems in different topics are proved and stored in collections used as a knowledge base for the theorem prover. By looking at the theorems and lemmas used in the proof of a previously verified theorem, a user may gain insight on how to steer a current proof attempt.

The hierarchy of theorems arranged in an order that shows interdependency forms a directed acyclic graph. This can be visualized using a node-link diagram as shown in figure 2.

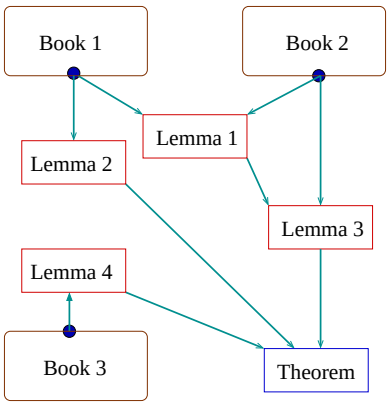

Figure 2: A schematic view of lemmas and books used to prove a theorem

In this paper, we will present details and results on visualization of the next two levels.

4.2 Proof tree visualization

Most proof attempts from theorem provers have a tree structured approach. The root of the tree can be considered to be the main theorem being proved. A theorem prover then either proves/disproves the theorem or divides the theorem into subgoals. Each of these subgoals is then tackled, in an order determined by the particular theorem proving system. ACL2 tends to use a depth first search.

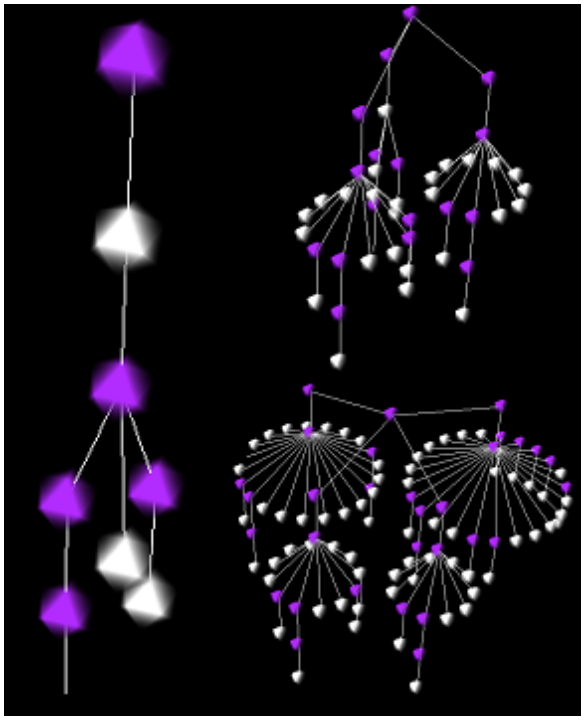

Figure 3: Proof tree visualization. Three different time steps during a proof attempt are shown in clockwise order.

Actions taken by the theorem provers like breaking up a complex expression into a composition of several sub-expressions, each of which can be proved in a similar fashion, yields very similar structures. Different structures in the proof tree provide different insights about the actions taken by the theorem prover at those points. It becomes easy to observe induction, or simplification being applied to a subgoal.

Another reason to perform proof tree visualization is to observe the structure of the proof, without the limitations placed in the case of textual data, and to navigate the proof tree more easily. The overall structure of the proof tree, combined with a close mapping to the text data, and vice versa, allows users to navigate failed proofs comfortably.

Additional visual information in proof trees There is a lot of information that can be clearly presented to the user through the visualization of proof trees. While the structure of the overall proof will be clear in the tree structure, visual cues are used to present interesting details.

- ACL2 has a model in which the subgoals can be reduced using generalization, induction, simplification etc. These actions are limited and distinct. The action taken by the theorem

prover at any subgoal can be visualized by the corresponding node's color (see figure 5).

- ACL2 provides the user with a set of statements indicating its reasoning at each subgoal. This text can be parsed and stored at each node of the proof tree to be accessed by the user.

- The subgoal expressions at a node are later used for expression visualization, but are also stored at this level. Textual and graphical multi-view visualization provides details through the text, and overall structure through the graphics at the same time. This is illustrated in figure 5.

- As the theorem prover proceeds, we have a set of nodes that were generated, a set of nodes that were proved, and possibly a node that could not be proved. Such information can be dynamically stored at the links by coloring them appropriately.

4.2.1 User interactions

The following interactions are considered necessary for interacting with a proof attempt's tree:

- The rendering of the proof tree must be synchronized with the progress of the proof attempt. This helps us see patterns developing as the proof proceeds. It becomes trivial to catch many infinite recursions, understand the structure of the proof, see the current part of the main theorem being tackled, locate those subgoals that were not proved immediately, and identify the inductions which were attempted. Figure 3 shows the progress of a proof attempt at three different time steps (in clockwise order).

- Rotating, zooming, and panning of the proof tree, without losing track of nodes of interest.

- The movement from one subgoal of interest to another must be smooth. Animation sequences must help the user navigate without losing the overall location of the viewpoint in the tree. To do this, we choose a unique path from the current node, to the root and then down to the other node.

- Users should be able to move from the text to the visual tree (and vice versa), by selecting subgoals in either.

- Some domain specific interactions required would be obtaining definitions and previous uses of axioms.

To ensure real-time user interaction with the multi-view visualization of the proof process, we need fast construction and rendering of the proof tree. Since proof attempts tend to be large, taking possibly hours to finish, users prefer being given synchronized feedback in both textual and graphical views of the current state of the proof. Thus we need to use a dynamic tree construction and rendering algorithm.

4.2.2 Dynamic tree construction

We get the information on the number of children at a node as soon as each node is evaluated. Hence we know the distribution of nodes at each level as the theorem prover starts proving them in order.

We use a variant of the cone tree layout algorithm to render our trees interactively.

The implementation costs $O(nlog(n))$, where n is the total number of nodes in the tree. It does not require pre-processing (which was important so that we could obtain synchronized interactive visualizations of information from the theorem prover).

4.3 Expression visualization

When a large complex proof fails, it is relatively easy to obtain the point of failure in the overall proof structure, by looking at the proof tree and with knowledge of the theorem being proved. The main hurdle in finding out why the theorem prover could not prove a theorem is understanding the critical node at which the theorem prover failed or deviated from the expected path.

Expressions representing the subgoal at a node can be arbitrarily large. The expression tree of such a function can be visualized as a two-dimensional tree.

We make the following observations:

- The proof proceeds by manipulating arguments, possibly through substitution, permutations, or using some rule to change the current formula to another, but retaining the arguments in some form.

- Different parts of the tree are usually related. This is because large proofs that are proved in similar ways tend to have similar expressions in them. The difference between such expressions could be of interest to the user.

- While the operation performed could be simple, the actual subtrees could be arbitrarily large. A simple operation in which two arguments to a function were swapped could be extremely difficult to catch by looking at text.

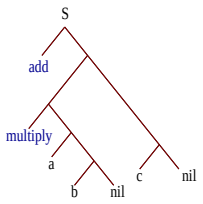

Figure 4: Function Visualization

Figure 4 is the tree of the LISP expression $(add\ (multiply\ a\ b)\ c)$ *add* has two parameters: $(multiply\ a\ b)$ and c. *add* and *multiply* are operators.

Need for tree matching In theorem proving, larger proof attempts are always cumbersome to follow. From one goal to another, the theorem prover performs some actions, modifying the expressions at each level. In order to follow changes, pattern matching can be applied to the expressions (after suitably representing them as trees). Users can use this pattern matching to see what is changing at each step and also verify that the proof is proceeding as expected.

Consider ACL2. In a typical step, it transforms a formula into a "simplified" formula using previously proved lemmas, definitions etc. The simplified formula is not necessarily smaller, but is generally closer to some canonical form. Typically, the user is familiar with the formula before simplification, having understood the preceeding steps in the proof attempt. ACL2 displays the formulas after simplification, together with a note listing the names of the lemmas and definitions used. For smaller formulas, it is not difficult for the user to understand what happened.

But, if the two formulas each required several megabytes to print, it is virtually impossible to comprehend the transformation. What is needed are ways to:

- abstract a large formula so that its structure can be comprehended

- use the screen real estate to make explicit the necessary details in the formula
- visually communicate how the formula is related to others

A key component of this problem is similarity matching, ie. quantifying how much one expression resembles another.

The representation of an expression as a binary tree is just the Lisp representation of expressions. Thus, in our case, we perform the approximate matching of trees. Consider any expression E. E can be represented as a function symbol F operating on a set of parameters $p_1, p_2, \ldots p_n$. In a binary tree representation, the left child of the root node contains the function symbol F. The parameters are then the left children of all the nodes on the right side path from the root to a leaf. See figure 4 for an example. While most tree matching algorithms try to match children of roots, we slightly change this to matching function symbols and parameters in the binary tree structure. The binary tree structure is maintained to help in parsing information from the parser to tree form and vice versa.

Heuristics for matching We define a similarity factor χ, bounded between 0 and 1, which is 1 if and only if the two trees being compared match exactly. Two trees which have different operators will result in a low value of χ. Through tree matching, we need to obtain close matches when two trees differ only at the leaves. This requires χ to reflect the first level at which the difference was present (note that level refers to level in the parameters and not the binary tree itself). Permutations are tougher to handle. We should allow for permutations in the subtrees in the sense that two trees which are exactly similar, allowing for permutation of parameters must be given a high value of χ.

We need to augment similarity with more quantitative information such as the depth and parameter number that has changed.

We use a simple domain specific heuristic based algorithm that can help prune the search space quickly. When the proofs proceed from one step to another, parameters could get swapped. Thus we need to do unordered matching. At each step we could use a depth threshold to further optimize the matching. This is consistent with our goal of achieving higher similarity when the higher levels in the tree match. As presented in section 3.2, there are many algorithms which improve on the simple quadratic bound for ordered searching, and polynomial time for unordered searching.

Running time of algorithm The algorithm we use is similar to the recursive algorithms presented by [Hoffmann and O'Donnell 1982] which take $O(mn)$ time, where $n = |T|$ and $m = |P|$ to obtain all ordered matches, and P is the pattern tree being searched in tree T. See [Luccio et al. 2001] for faster unordered matching algorithms. Using a depth threshold, we can reduce this complexity, but get worse approximations.

5 Synchronized Multi-view Information Visualization

Our environment provides synchronized multi-view representations of the proof tree to the user. The pretty printed text is synchronized with our graphical visualization presenting the information in two different formats enabling better comprehension.

The visualization helps the user identify similarities, browse the entire proof expediently, and ascertain location information about a particular subgoal. The user still needs to read the text for details. The point of failure of a proof attempt can be easily identified through graphical visualization.

We can achieve synchronization by updating users actions in one view, in the other. Browsing and searching can be updated in both views simultaneously. Figure 5 is a multi-view representation of a particular proof attempt. The text windows on the left contain the contents of some nodes from the proof tree on the right.

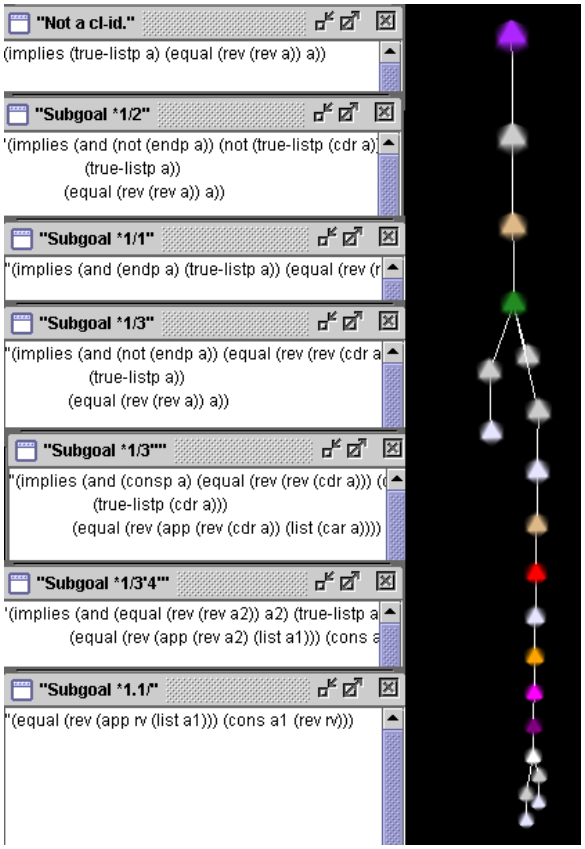

Figure 5: Multi-view text and graphical visualization of a proof attempt. The text windows on the left contain the contents of some nodes from the proof tree on the right.

Hierarchical visualization As proof trees or functions in subgoals get larger, there needs to be some hierarchical representation for the visualizations to be effective. This needs to be done in both the textual and graphical domain for two reasons. First, the text can easily exceed hundreds of pages, and the number of nodes rendered in the tree can be very large. Second, we are trying to achieve a synchronized multi-view representation. Graphical tree representations have generally created hierarchies by collapsing subtrees into different models, and we use the same approach in our case. For textual visualizations, one of the ways in which hierarchical representation can be managed is through hyperlinks. Functions can be replaced by symbols to form a hierarchy, in synchronization with the graphical tree. In [Megill 2002], we can see examples of how proofs can be presented in one hierarchical manner. Multi-resolution techniques for rendering text could be more useful to browse than hypertext and needs to be further examined.

6 Case studies

We have implemented the dynamic proof tree visualization component, with a Java client. In figure 3, we see a sample proof tree visualization, with node colors representing the type of action taken by the theorem prover at that stage. This was a proof involving threads, states and state transitions, and proving the correctness of

a state in the Java Virtual Machine. It is an example of a large practical proof, involving 1772 subgoals. The primary goal is broken up into 81 subgoals. The output from the proof attempt is 364,433 lines of text (about 13.6 Mb). The rendering of the proof tree is synchronized with the progress of the theorem prover.

The visualization of the results from the pattern matching been implemented in both text and graphics. In figure 6, we see two sets of texts (in the middle and right column). A sub-expression from column two is matched with the entire expression on the right. The values returned by χ were used to color the text. The font color indicates how similar an expression is to the search expression. Unselected text is light grey, while selected text is black. The results from pattern matching are shown by varying the font color from bright red (high match) to dark red (low match).

The results from the pattern matching are also shown with the synchronized graphical expression viewer. The result is shown in figure 6 (in column on the left). The unselected sections are grey, while selections are cyan. The selected expression in the top tree in the left column was searched for in the selected part of the middle tree. Again, bright to dark red is used to show high to low matches between the patterns. The third tree in the left column is a zoomed in view of the outlined box in the second tree.

Figure 5 is a screen shot from the proof of the proposition that the reverse of the reverse of a list is the list itself (given certain conditions and definitions). The proof is a canonical example since the subgoals are proved using a variety of different methods.

7 Conclusion

We provide an approach to formal methods visualization. We identify a hierarchy of levels where graphical visualization will augment the text interface to help users better understand the proof process. We provide details of a system where a synchronized interactive symbolic visualization through a graphical user interface helps users navigate, browse and view animated dynamic views of the proofs. We also include in our system a pattern matching component to help users debug proofs by following and comparing changes in subgoals.

8 Acknowledgments

We are grateful to Robert Krug for writing the socket code that helps us communicate with ACL2. Research supported in part by grants from NSF CCR-9988357and ACI 9982297.

References

B. BROCK, M. KAUFMANN, AND JS. MOORE. 1996. ACL2 theorems about commercial microprocessors. In *proceedings of first international conference on Formal Methods in Computer-aided Design*, Springer Verlag, Palo Alto, CA, USA, M. Srivas and A. Camilleri, Eds., vol. 1166, 275–293.

BOYER, R. S., AND MOORE, J. S. 1988. *A computational logic handbook*. Academic Press Professional, Inc.

BROWN, M. H. 1987. *Algorithm animation*. MIT press.

BUTLER, R. W., MILLER, S. P., POTTS, J. N., AND CARRENO, V. A. 1998. A formal methods approach to the analysis of mode confusion. In *proceedings of 17th AIAA/IEEE Digital Avionics Systems Conference*.

CARRIERE, J., AND KAZMAN, R. 1995. Interacting with huge hierarchies: Beyond cone trees. In *proceedings of IEEE Information Visualization*, 74–78.

COLE, R., AND HARIHARAN, R. 1997. Tree pattern matching and subset matching in randomized o(nlog3m) time. In *proceedings of the twenty-ninth annual ACM symposium on Theory of Computing*, ACM Press, 66–75.

COLE, R., HARIHARAN, R., AND INDYK, P. 1999. Tree pattern matching and subset matching in deterministic o(n log3 n)-time. In *proceedings of the tenth annual ACM-SIAM symposium on Discrete Algorithms*, ACM Press, 245–254.

DUBINER, M., GALIL, Z., AND MAGEN, E. 1994. Faster tree pattern matching. *Journal of the ACM (JACM) 41*, 2, 205–213.

FRANK VAN HAM, HUUB VAN DE WETERING, J. J. V. W. 2001. Visualization of state transition graphs. In *proceedings of IEEE Symposium on Information Visualization*, 59–66.

GOGUEN, J. A. 1999. Social and semiotic analyses for theorem prover user interface design. *Formal Aspects of Computing 11*, 3, 272–301.

GORDON, M. J. C., AND MELHAM, T. F. 1993. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

HOFFMANN, C. M., AND O'DONNELL, M. J. 1982. Pattern matching in trees. *Journal of the ACM (JACM) 29*, 1, 68–95.

JEONG, C. S., AND PANG, A. 1998. Reconfigurable disc trees for visualizing large hierarchical information space. In *proceedings of IEEE Information Visualization*, Computer Society Press, 19–25.

KAUFMANN, M., AND MOORE, J. S. 1994. Design goals of ACL2. Tech. Rep. 101, Computational Logic, Inc., August.

KAUFMANN, M., AND MOORE, J. S. 1997. An industrial strength theorem prover for a logic based on common Lisp. *Transactions on Software Engineering 23*, 4, 203–213.

KELLER, P. R., AND KELLER, M. M. 1994. *Visual cues: Practical data visualization*. IEEE Computer Society Press.

KOIKE, H., AND YOSHIHARA, H. 1993. Fractal approaches for visualizing huge hierarchies. In *Proceedings of IEEE Symposium on Visual Languages, VL*, IEEE Computer Society, E. P. Glinert and K. A. Olsen, Eds., 55–60.

KOSARAJU, S. R. 1989. Efficient tree pattern matching. In *proceedings of 30th Annual Symposium on Foundations of Computer Science*, 178–183.

LAMPING, J., AND RAO, R. 1994. Laying out and visualizing large trees using a hyperbolic space. In *proceedings of the 7th annual ACM symposium on User Interface Software and Technology*, ACM Press, 13–14.

LUCCIO, F., ENRIQUEZ, A. M., RIEUMONT, P. O., AND PAGLI, L. 2001. Exact rooted subtree matching in sublinear time. Tech. Rep. TR-01-14, University of Pisa, 09.

MEGILL, N. D., 2002. Metamath : A computer language for pure mathematicians. http://metamath.org.

MUNZNER, T. 1998. Exploring large graphs in 3d hyperbolic space. *IEEE Computer Graphics and Applications 18*, 1 (July/Aug), 18–23.

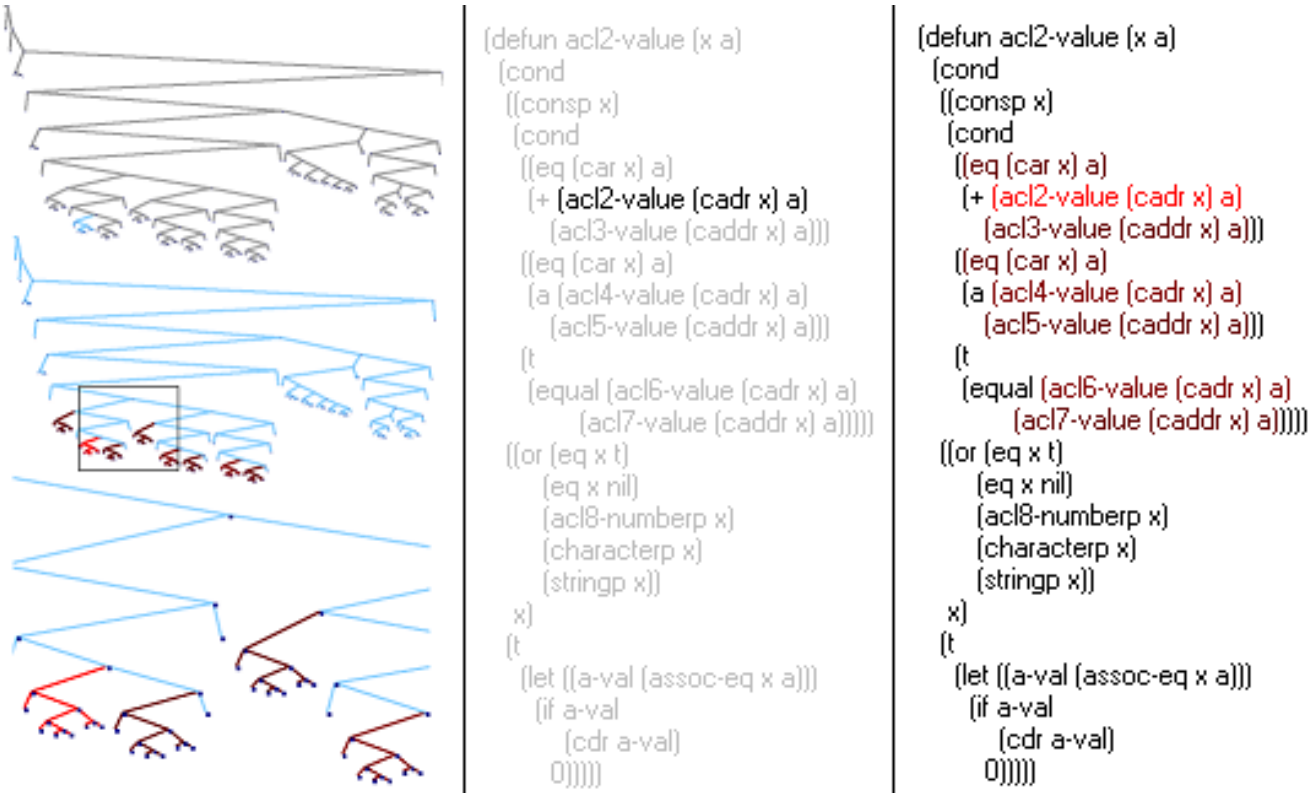

Figure 6: A synchronized view of text and graphics visualizations from level 3. Pattern matching of expressions from a proof: A composition of screen shots from our implementation.

OWRE, S., RUSHBY, J. M., AND SHANKAR, N. 1992. PVS: A prototype verification system. In *proceedings of 11th International Conference on Automated Deduction (CADE)*, Springer-Verlag, Saratoga, NY, D. Kapur, Ed., vol. 607 of *Lecture Notes in Artificial Intelligence*, 748–752.

REINGOLD, E. M., AND TILFORD, J. S. 1981. Tidier drawings of trees. *IEEE Transactions on Software Engineering 7*, 2, 223–228.

ROBERTSON, G. G., MACKINLAY, J. D., AND CARD, S. K. 1991. Cone trees: animated 3D visualizations of hierarchical information. In *proceedings of the SIGCHI conference on Human Factors in Computing Systems*, ACM Press, 189–194.

SHNEIDERMAN, B. 1992. Tree visualization with tree-maps: 2-D space-filling approach. *ACM Transactions on Graphics (TOG) 11*, 1, 92–99.

THIRY, L., BERTOT, Y., AND KAHN, G. 1992. Real theorem provers deserve real user-interfaces. In *proceedings of the fifth ACM SIGSOFT symposium on Software Development Environments*, ACM Press, 120–129.

TUFTE, E. 1990. *Envisioning information*. Graphics Press, Cheshire, Connecticut.

WALKER, J. Q. 1990. A node-positioning algorithm for general trees. *Software Practice and Experience 20*, 7, 685–705.