


The system has been designed for developing large interactive proofs. In particular, the GUI provides commands for reading and writing hierarchical proofs by letting the user focus on part of a proof. TLAPS uses a fingerprinting mechanism to store proof obligations and their status. It thus avoids reproving previously proved obligations, even after a model or a proof has been restructured, and it facilitates the analysis of what parts of a proof are affected by changes in the model.

The paper is a longer version of an article published at FM 2012.

### 3.27 Case Based Specifications – reusing specifications, programs and proofs

*Rosemary Monahan (Nat. University of Ireland, IE)*

License  Creative Commons BY-NC-ND 3.0 Unported license

© Rosemary Monahan

Joint work of Monahan, Rosemary; O'Donoghue, Diarmuid

URL <http://www.cs.nuim.ie/staff/rosemary>

URL <http://www.cs.nuim.ie/staff/dod>

Many software verification tools use the design-by-contract approach to annotate programs with assertions so that tools, such as compilers, can generate the proof obligations required to verify that a program satisfies its specification. Theorem provers and SMT solvers are then used to, often automatically, discharge the proof obligations that have been generated.

While verification tools are becoming more powerful and more popular, the major difficulties facing their users concern learning how to interact efficiently with these tools. These issues include learning how to write good assertions so that the specification expresses what the program must achieve and writing good implementations so that the program verification is easily achieved [4, 5]. In this presentation we discuss guiding the user in these aspects by making use of verifications from previously written programs. That is by finding a similar or analogous program to the one under development, we can apply the same implementation and specification approaches. Our strategy is to use a graph-based representation of a program and its specification as the basis for identifying similar programs.

Graph-matching was identified as the key to elucidating analogical comparisons in the seminal work on Structure Mapping Theory [1]. By representing two sets of information as relational graphs, structure mapping allows us to generate the detailed comparison between the two concepts involved. So given two graphs we can identify the detailed comparison using graph matching algorithms. For one application of graph matching to process geographic and spatial data see [3]. However, we may not always have an identified “source” to apply to our given problem. Thus, more recent work has taken a problem description, searching through a number of potentially analogous descriptions, to identify the most similar past solution to that problem [2].

Our work will develop a graph matching framework for program verification. The associated tools will operate on a collection of previously verified programs, identifying specifications that are similar to those under development. The program associated with this “matching specification” will guide the programmer to construct a program that can be verified as correct with respect to the given specification. Likewise, the strategy can be applied when the starting point is a program for which we need to construct a correct specification.

The core matching process can be thought of as a K+J colored graph-matching algorithm,

which coupled with analogical transfer will re-apply the old solution to a new problem. Graphs can be flow graphs, UML diagrams, parse trees or another representation of a specification. Therefore, an iterative implementation of a sigma function (say) using tail recursion will be analogous to another recursive implementation—possibly using head-recursion. Similarly, iterative calculations of the same function using while and for loops will be more analogous to one another. Identical graph matching (isomorphism) will identify exact matches, given the representation, while non-identical (homomorphic) matches will identify the best available solutions.

In summary, our work will help to make software specification and verification more accessible to programmers by guiding users with knowledge of previously verified programs. A graphical representation of the specification, coupled with graph matching algorithms, is used as the basis of an analogical approach to support reuse of specification strategies.

### References

- 1 D. Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.
- 2 D. P. O’Donoghue and M. T. Keane. A creative analogy machine: Results and challenges. In M. L. Maher, K. Hammond, A. Pease, R. Pérez y Pérez, D. Ventura, and G. Wiggins, editors, *4th International Conference on Computational Creativity (ICCC 2012)*, pages 17–24, 2012.
- 3 D. P. O’Donoghue, A. J. Bohan, and M. T. Keane. Seeing things: Inventive reasoning with geometric analogies and topographic maps. *New Generation Comput.*, 24(3):267–288, 2006.
- 4 K. R. M. Leino and R. Monahan. Automatic verification of textbook programs that use comprehensions. In *9th Workshop on Formal Techniques for Java-like Programs (FTfJP 2007)*, *ECOOP 2007 Workshop*, Berlin, Germany, July 2007.
- 5 K. R. M. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In G. T. Leavens, P. W. O’Hearn, and S. K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *LNCS*, pages 112–126. Springer, 2010.

## 3.28 Can AI Help ACL2?

*J Strother Moore (University of Texas at Austin, US)*

**License** © ⓘ ⊖ Creative Commons BY-NC-ND 3.0 Unported license  
© J Strother Moore

**Joint work of** Moore, J Strother; Kaufmann, Matt; Boyer, Robert

**Main reference** M. Kaufmann, P. Manolios, J. S. Moore, “*Computer-Aided Reasoning: An Approach*,” volume 3 of *Advances in Formal Methods*. Kluwer Academic Publishers, 2000.

**URL** <http://www.cs.utexas.edu/users/moore/acl2>

ACL2 stands for “A Computational Logic for Applicative Common Lisp,” and is a fully integrated verification environment for functional Common Lisp. I briefly mentioned some of its industrial applications, primarily in microprocessor design, especially floating-point unit design, and security. ACL2 is used to prove functional correctness of industrial designs. I then demonstrated an ACL2 model of the Java Virtual Machine highlighting (a) the size and scale of the formal model, (b) the fact that it was executable and thus was a JVM engine, and (c) ACL2 can be configured so that code proofs are often automatic. I then turned to how AI could help the ACL2 user, including: facilitating proof maintenance in the face of continued evolution of designs; facilitating team interaction in team based proofs (e.g., automatically informing team member A that team member B has already proved a lemma that seems