Abstract—This paper describes recent search efficiency enhancements in an integrated automated prover built with the goal of efficiently verifying verification conditions that are obviously valid. Such conditions arise from showing code conformance to specifications using an easily expandable library of reusable mathematical theorems.

I. INTRODUCTION

The tool described in this paper is an integral part of a verifying compiler. Its purpose is to provide feedback on program correctness, and has been used successfully as a learning aid in undergraduate software engineering courses [1]. The verification condition prover implements a decision procedure for the theory of equality and applies universally quantified assertions housed in a static library of theorems.

II. CONGRUENCE KEEPER

The congruence keeper is essentially a union-find [2] data structure for logical expressions without quantifiers. This is achieved by maintaining a union-find data structure for a set of symbols, and associating these symbols with function applications. The congruence keeper contains two kinds of nodes. A symbol node contains a field for a symbol, one link to a parent symbol node, and one or more links to expression nodes. These expression node links are categorized by position.

Expression nodes represent function applications of at least one argument. They are essentially unique combinations of links to symbol nodes. An expression node contains one link to the symbol node which is the canonical representative of the class of things equal to it, and one link to a symbol node for each position in the expression. Typically an expression node would contain a data field for the operator. However, problems would occur when functions are used as arguments to functions, as the algorithm would need to handle changing multiple copies of function symbols. Maintaining the expression nodes as unique combinations of symbols allows for quick merging of function operators. Though quantifiers are not supported in the congruence keeper, it is used in tandem with a theorem library which may contain expressions in higher order logic, and so support for efficiently searching for operators used as arguments, and possibly merging these operators with others is necessary.

The congruence keeper has been recently augmented with a system that allows for quickly finding the usages of symbols by position. These are the “argument links” extending from the symbol nodes in the center area as shown in Figure 1. This is useful for wild card searches, a part of the theorem matching process. Another improvement is the addition of integrated logical axioms for $\land$, $\lor$, and $=$. Axioms may be applied to each new expression node having a logical operator. For example, if the node entered reads as “$p \lor q$”, and this node is congruent to false, then $p$ and $q$ are queued to be merged with false.

Figure 1 represents the initial state of a proof of $I + J \leq J + K \rightarrow I \leq K$. We have symbol nodes for each symbol used, with the addition of true, false, and three created symbols: _G, _C1, and _C2. Initially, each symbol is known to be equal only to itself, and so the parent links point back to their origin. It can happen that the expression nodes take on multiple interpretations, so they are simply labeled with letters. Expression nodes in Figure 1 correspond to the internal nodes in Figure 2. Labels A - D show the correspondence between the nodes. For example, Node A comes from the goal statement. It represents $I \leq K$, and is congruent to node _G. The proof concludes when there is a parent link path from either node _G or the false node to the true node.

---

III. THEOREM APPLICATIONS

We apply theorems from the library by first splitting them into premise and conclusion sections. Implications are treated as-is, while equations are treated as bi-implications. If the matcher finds a fit between a premise and the antecedent of
the verification condition, the conclusion is marked as a valid candidate for entry.

The previous version ignored cases where a quantified symbol appeared in the conclusion, but not the premise. Now this is handled by doing a non-disqualifying search on fragments of the partially instantiated conclusion. Support has also been added for transitivity of implication. We look for instances of the conclusion of a theorem in the goal of the verification condition. If found, we can add a new goal, which is the premise of that theorem.

IV. PLANNED IMPROVEMENTS

While the searching and congruence closure operations are currently handled efficiently, we need a way to retain previous search results, otherwise the program must start each search fresh and will consequently end up repeating work it has already accomplished. Rather than iterating over theorems and looking for matches among the assertions that make up the VC as the program does currently, the planned new version will iterate over the VC assertions and look for ways the assertion can fit into a structure that represents the theorem library. The advantage of this new system is that each unique VC assertion need only be evaluated once, while the old system often requires looping through the theorems several times.

ACKNOWLEDGMENTS

This research is funded in part by NSF grants CCF-1161916 and DUE-1022941. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF. Thanks to Bill Ogden, Joan Krone, and Murali Sitaraman for their contributions to this paper.

REFERENCES
