experiment with this implementation measures the size of differences and the time taken to compute it.

The test subjects for this experiment include a collection of small programs, a selection of medium programs, and one "larger" program. Their sizes are in the hundreds, thousands, and tens of thousands of lines, respectively. Presently data for the first group has been collected.

These programs, which were obtained from Siemens Corporate Research, simulate small changes designed to be realistic (neither too easy nor too difficult to detect).

Each programs has between 7 and 21 functions and ranges from 145 to 514 non-blank lines of source code. Each program has from 7 to 42 different versions.

Consecutive pairs of versions were used as the previously tested program and the modified version of this program as input into the implementation.

The table below reports the average, best case, and worst case size reduction for each program over all its versions.

The initial data look promising. Most small programs are single thought by their very nature. For such programs, differences is expected to include most of modified.

Referring to the table below, the worst case reduction is 9%, while the best is 95%. Over all the versions the average reduction was 26%.

For these short programs no measurable time was taken in computing differences. With the exception of the program replace, there is an upward trend in the average reduction obtained as the program size increases. Further experiment will reveal if this trend continues.

program name	number of versions	size	percent size reduction worst	average	best
calc	11	166	9%	15%	45%
print_t1	7	513	38	39	39
print_t2	10	468	33	34	38
replace	32	514	11	12	19
schedule1	9	362	27	27	28
schedule2	9	280	19	33	95
tcas	40	145	17	24	27
AVERAGES	17	350	22	26	42

A Fully Capable Bidirectional Debugger[5]

Bob Boothe
University of Southern Maine
Portland, Maine
(boothe@cs.usm.maine.edu)

Introduction

The goal of this research project is to develop a bidirectional program debugger with which one can move as easily backwards as current debuggers move forward. We believe this will be a vastly more useful debugger. A programmer will be able to start at the manifestation of a bug and proceed backwards investigating how the program arrived at the incorrect state, rather than the current and often tedious practice of the user stepping and breakpointing monotonically forward and then being forced to start over from the beginning if they skip past a point of interest.

Our experimental debugger has been implemented to work with C and C++ programs on Digital/Compaq Alpha based UNIX workstations.

Techniques

We have abandoned the traditional debugger implementation technique of dynamically inserting trap instructions at potential stopping points in the program being debugged. In its place we have developed a technique which uses a collection of embedded counter routines to track the progress of the program and stop it precisely at the final target location. While these embedded counters add some overhead (less than a factor of 2), they allow us to efficiently move backwards to earlier points in the execution by re-executing the program and stopping at earlier counter values.

The basic counters are the "step counter" and "call depth counter." These are inserted at compile time when the program is compiled for debugging. Calls to the step counter are inserted at the traditional debugger stepping points: each line starting a new statement. When the user inserts a breakpoint, we dynamically replace the call to the step counter at the breakpoint location with a call to the breakpoint counter. The step counter and breakpoint counter allow us to locate and stop at any specified number of steps or breakpoints in either the forward or reverse direction. More complex movements such as "next" and "finish," and their backwards analogues "previous" and "before," use the call depth counter along with specialized counter routines that replace the basic step counter. Finally, we have implemented efficient "until" and "back until" movements that proceed forward or backwards until a specified variable either changes or reaches a desired value.

To provide efficient backwards movements in long running programs, we create periodic checkpoints, so that re-execution need only execute forward from the nearest preceding check-

[5]This research was partly supported by NSF grant CCR-9619456.

point, and to provide deterministic re-execution, we provide I/O logging to capture external inputs from the initial execution that we will replay later upon re-execution.

Future Directions

So far we have focused our efforts on solving the most challenging research problems of providing a full set of debugging movement commands that operate efficiently in both the forward and a backwards directions. We have tested this extensively and are exceptionally pleased with its performance, but we have not yet unleashed it upon our students for general use in debugging their programs. This is the ultimate test: whether students find the backward movements helpful in more quickly locating the causes of bugs in their programs. We are currently working on user interface level issues in preparation for this next level of evaluation.

A Composite Model Checking Toolset for Analyzing Software Systems

Tevfik Bultan
University of California
Santa Barbara, CA 93106
(bultan@cs.ucsb.edu)

Model checking has proved to be a successful technique for verifying hardware systems. Given a transition system and a temporal property, model checking procedures exhaustively search the state space of the input transition system to find out if it satisfies the given temporal property. Recently, model checking has been used for analyzing software specifications with encouraging results [CAB+98]. The state-space of a software specification can be explored using model checking procedures to verify or falsify (by generating counter-example behaviors) its properties.

The success of model checking has been partially due to Binary Decision Diagrams (BDDs) – a data structure that can encode boolean functions in a highly compact format. The main idea in BDD based model checking is to represent sets of system states and transitions as boolean formulas, and manipulate them efficiently using BDDs [McM93]. BDD data structure supports the operations required for model checking: intersection, union, complement, equivalence checking and existential quantifier elimination (used to compute pre- and post-conditions). This type of model checking is called *symbolic* since the system states are represented implicitly by BDDs during the state space search.

In recent years new symbolic representations have been proposed. For example, HyTech, a symbolic model checker for hybrid systems, encodes real domains using linear constraints on reals [AHH96]. Recently, we proposed a model checker for integer based systems, which uses Presburger arithmetic (integer arithmetic without multiplication) constraints as its underlying state representation [BGP97]. Using constraint representations one can verify systems with infinite variable domains (which is not possible using finite representations such as BDDs).

Our goal in this project is to develop a toolset which combines various symbolic representations in a single composite model checker. In the composite model checking approach each variable in the input system is mapped to a symbolic representation type [BGL98]. (For example, boolean and enumerated variables can be mapped to BDD representation, and integers can be mapped to Presburger constraint representation.) Then, each atomic event in the input system is conjunctively partitioned where each conjunct specifies the effect of the event on the variables mapped to a single symbolic representation. Conjunctive partitioning of the atomic events allows pre- and post-condition computations to distribute over different symbolic representations.

We plan to structure the composite model checking toolset using a layered class hierarchy. The lowest layer will contain libraries for manipulating various symbolic representations such as BDDs and arithmetic constraints. We plan to develop an API which will be shared by different symbolic representations. At the next level of the hierarchy we will have the composite-model library to handle operations over mixed-type expressions (e.g., equivalence check, intersection, etc.); in turn, these operations will invoke their relevant type-specific counterparts in the lower level to help carry out the desired effect. At the top level, the model checker will implement the fixpoint computations using the composite-model library. We already implemented a prototype toolset based on this structure which combines BDD and Presburger constraint representations [BGL98]. We plan to expand our composite model checker by adding other symbolic representations which will allow us to encode variable types such as reals and queues. We would also like to compare performances of different symbolic representations.

We plan to investigate techniques for generating efficient symbolic representations for software specifications. Particularly, we would like to investigate automated or semi-automated techniques for abstraction, partitioning, and compositional analysis. Our goal is to use the composite model checking toolset to investigate effectiveness of symbolic analysis techniques in verification of software systems.

References

[AHH96] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181– 201, 1996.

[BGL98] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 International Symposium on Software Testing and Analysis*, pages 113–123, 1998.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 400–411. Springer, 1997.

[CAB+98] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software