Cleanly combining specialised program analysers

Nathaniel Charlton
nac103@doc.ic.ac.uk

Michael Huth
mrh@doc.ic.ac.uk

Abstract

Automatically proving that (infinite-state) software programs satisfy a specification is an important task, but has proved very difficult. Thus, in order to obtain techniques that work with reasonable speed and without user guidance, researchers have typically targeted restricted classes of language features, programming idioms and properties. We have designed a system in which several of these specialised techniques can be used together in proving that a program is correct; this is done without breaking modularity by propagating information between the analyses, expressed as formulae of an expressive common logic. In this way, we can verify programs which, because they use diverse language features and idioms, are difficult or impossible to prove using any one individual technique. Our system is implemented in the experimental tool HECTOR.

1 Overview

Automatically proving that (infinite-state) software programs satisfy their behavioural specifications has been a goal of computer science for many years, and is of both practical and mathematical interest. Software is more pervasive and safety-critical than ever, and conventional testing techniques continue to miss significant flaws. To this end, many algorithms have been developed for reasoning about the sets of states a program may reach.

However, proving program correctness has proved very difficult. One issue we mention is that adding transitive closure to logics tends to make reasoning very difficult, yet some kind of transitive closure reasoning is needed to account for linked data structures such as linked lists and trees. Thus, in order to obtain techniques that work with reasonable speed and with an acceptable level of user guidance, researchers have typically targeted restricted programming languages (or certain programming idioms) and/or restricted kinds of behavioural properties. For example, some systems deal only with numerical relationships between simple variables, some only with the topology of linked data structures in the heap, and some consider only the way that Java-style objects access each others' fields. Abstract Interpretation [4] is arguably the dominant methodology for the development of such analysers.

We have designed a system in which several of these specialised techniques can be used together in proving that a program is correct. In this way, we can verify programs which, because they use diverse language features and idioms, are difficult or impossible to prove using any one individual technique. To make this possible, we allow the various analysers to exchange information about possible program states as they run. This information is expressed using formulae of a common logic, which includes arithmetic, transitive closure and first order quantification, but no second order constructs. By structuring our system this way, we allow the analysers to cooperate but do not break modularity: the writer of a new analyser only has to make his software understand the common logic, and it can then automatically work together with the existing software. Our work can be seen as being similar to the open product operator [3] (which inspired us) and the well-known Nelson-Oppen method [5]; but we allow arbitrary constraints from an expressive logic to be propagated, rather than merely simple queries or equalities.

The formal basis for our system was developed in [1], and our tool HECTOR [2] provides an experimental implementation. HECTOR analyses heap-manipulating imperative programs with recursion, and can be used online at `http://www.doc.ic.ac.uk/~nac103/hector/`.

2 Example

Figure 1 shows HECTOR running on a program manipulating linked lists of integers, and in particular the statement `y.next := x`. Each abstract program state (inner box) contains constraints from three analysers: (right to left)

1. *Predicate abstraction* uses a postcondition generator and a first order theorem prover to track the status of a vector of (user chosen) predicates. Arithmetic is handled properly but transitive closure formulae are treated as uninterpreted predicates.

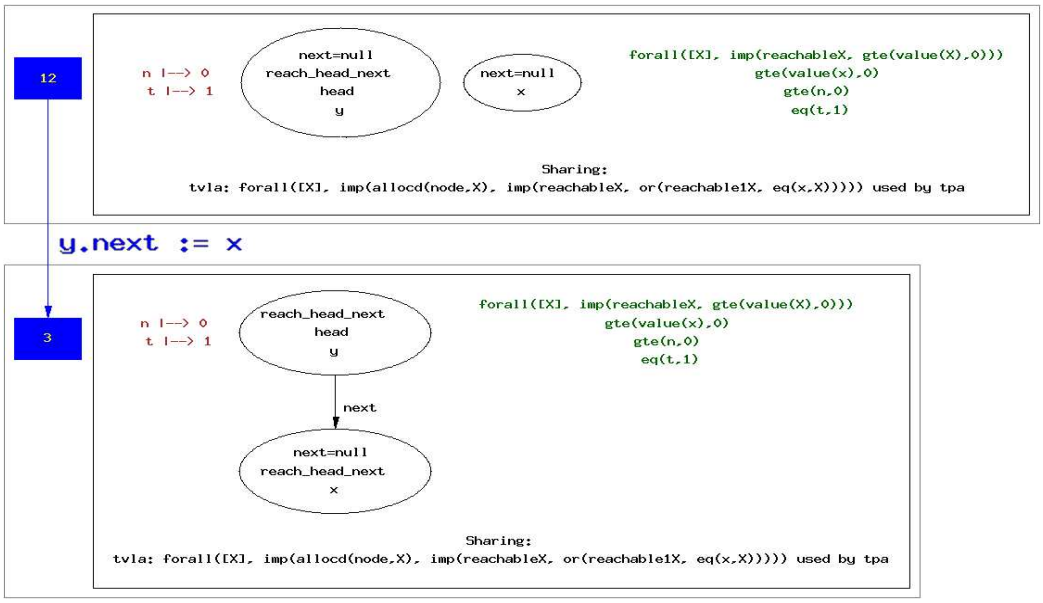

Figure 1: HECTOR uses a combination of three analysers to scrutinise the statement `y.next := x` in a program manipulating linked lists of integers.

2. *Three-valued shape analysis* [6] represents sets of heaps as models of three-valued logic, exploiting *summary nodes* and the third truth value *unknown* to generalise from a number of concrete heaps to the general "shape" they follow. It can perform transitive closure (reachability) reasoning about data structures, but ignores the integer data fields.
3. *Constant propagation* discovers variables which take a constant integer value through parts of the program; it is very shallow but runs very quickly.

In the figure, the three-valued shape analyser (which cannot handle arithmetic) propagates the following formula containing reachability information:

$$\forall X \left(\mathit{allocd}(\mathit{node}, X) \rightarrow (\mathit{reachable}(X) \rightarrow (\mathit{reachable}_1(X) \vee x = X)) \right) \tag{1}$$

where the subscript 1 means "in the previous state" and $\mathit{reachable}(X)$ is shorthand for the transitive closure formula

$$\mathit{allocd}(\mathit{node}, X) \wedge \left(X = \mathit{head} \vee \mathrm{TC}_{[A,B]} \left[\mathit{allocd}(\mathit{node}, A) \wedge \mathit{allocd}(\mathit{node}, B) \wedge \mathit{next}(A) = B \right] (\mathit{head}, X) \right)$$

and then the predicate abstraction analyser (which can reason about arithmetic but not reachability) makes use of (1) to obtain the necessary results.

References

[1] Nathaniel Charlton. Program verification with interacting analysis plugins. *Formal Aspects of Computing*, 2006. Springer Verlag, DOI: 10.1007/s00165-007-0029-4.

[2] Nathaniel Charlton and Michael Huth. Hector: software model checking with cooperating analysis plugins. To appear in Proceedings of *Computer Aided Verification* (CAV) 2007.

[3] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, August 2000.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[5] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[6] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.