Hector: Software Model Checking with Cooperating Analysis Plugins
(Tool Paper)

Nathaniel Charlton and Michael Huth

Department of Computing, Imperial College London
{nac103,M.Huth}@doc.imperial.ac.uk

Abstract. We present HECTOR, a software tool for combining different abstraction methods to extract sound models of heap-manipulating imperative programs with recursion. Extracted models may be explored visually and model checked with a wide range of "propositional" temporal logic safety properties, where "propositions" are formulae of a first order logic with transitive closure and arithmetic ($\mathcal{L}$). HECTOR uses techniques initiated in [4,5] to wrap up different abstraction methods as modular *analysis plugins*, and to exchange information about program state between plugins through formulae of $\mathcal{L}$. This approach aims to achieve both (apparently conflicting) advantages of increased precision and modularity. When checking safety properties containing non-independent "propositions", our model checking algorithm gives greater precision than a naïve three-valued one since it maintains some dependencies.

1 Introduction

Abstraction has been proposed as the key to building systems which automatically verify the correctness of software programs. Software written in everyday languages such as Java typically has an enormous or infinite state space, but abstraction can reduce this to a finite space of (abstract) states. The software verification literature describes many abstraction methods; predicate abstraction [2] and three-valued shape analysis [9] are two important examples.

This paper presents HECTOR, a Prolog implementation of abstraction-based verification which allows users to experiment with three interesting new features:

F#1 Abstractions are pluggable: Abstraction methods are wrapped inside modules called *analysis plugins*, which implement a common interface; algorithms for constructing and checking models are generic and use whatever plugins are activated. By activating several analyses together we can verify programs with diverse behavior, e.g. those which use both linked data structures and arithmetic. Such programs may be beyond the reach of shape analysis (which lacks a systematic treatment of arithmetic), and also beyond the reach of predicate abstraction (which doesn't handle linked data structures well). But the combination of the two analyses may succeed. The modular structure of HECTOR makes it easy to integrate and investigate new analyses under such cooperation.

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 168–172, 2007.

F#2 Plugins exchange information: Crucially, the interface which plugins implement allows them to exchange information about program state, expressed as formulae of a common logic $\mathcal{L}$. This information flow between the various plugins increases the precision of their respective analyses. Because there is a single common language, modularity is not broken. The implementor of a new abstraction method only has to make his plugin "understand" the common language, and the plugin will then automatically cooperate with existing ones.

F#3 Ad-hoc model checking: Because the abstraction process is generally costly, we maximize the utility of each abstract model generated by allowing the user to model check it with any property from an expressive safety language: the LTL fragment from [10], but where "propositions" are now any constraints on the program's current and initial states written in $\mathcal{L}$. Thus one can check for the absence of memory errors and assertion violations, but also much more.

The first two features were proposed and discussed in [4,5], where their formal basis is set out. We also refer to [4,5] for an account of related work. HECTOR's online version [6] can be used with plugins for monomial predicate abstraction, trivector predicate abstraction, three-valued shape analysis and constant propagation. (The predicate abstraction plugins call the theorem prover Simplify [11] and the shape analysis plugin calls the shape analyzer TVLA [12].) The web version offers some example programs to demonstrate various aspects; alternatively users may experiment with programs of their own.

2 Functionality of Hector

HECTOR maintains a list of models of (possibly different) programs; at any time users may build a new model, or select an existing one to draw graphically, model check, annotate with comments or delete.

2.1 Model Construction

To create a model, there are two steps.

S1 Enter the program to be analyzed: HECTOR analyses imperative, object-based programs, input as textual representations of control flow graphs (CFGs); this input language is sufficient to naturally express many simple Java-style programs. We currently don't handle inheritance, exceptions or concurrency.

S2 Configure the analyses: Firstly the user chooses which plugins to use. Secondly, the user configures each plugin; the settings for a plugin may tune the analysis it performs as well as how much information it propagates to other plugins. For the predicate abstraction plugins, for instance, a configuration consists of a choice of which abstraction predicates to use.

HECTOR then builds an abstract transition system which over-approximates the program, using a work-list algorithm which calls the selected plugins. Each abstract state is a tuple, containing one abstract value for each of the plugins; these are interpreted conjunctively. Recursion is handled by summarization,

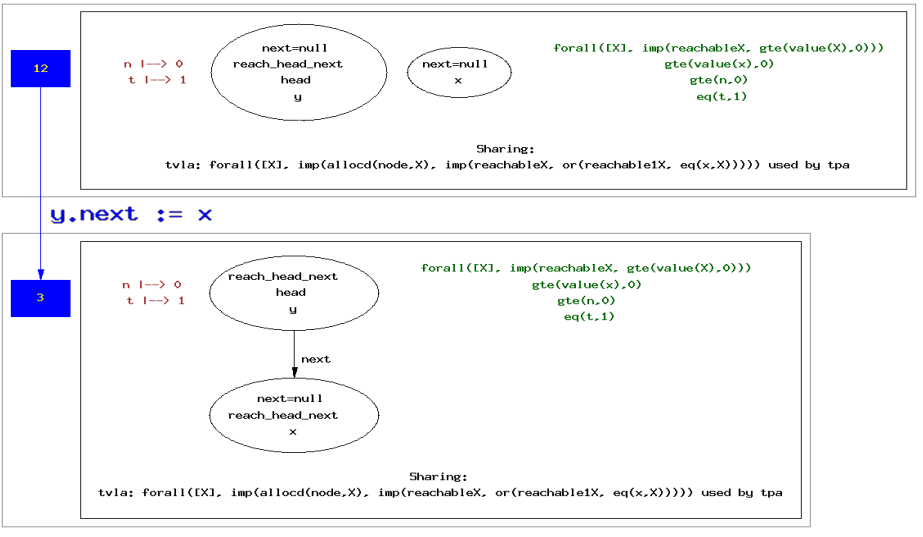

Fig. 1. Part of an abstract model, as generated and drawn by HECTOR. Program variables n and t are used for integers, *head*, x and y for addresses of list nodes.

as in [1]. As stated earlier, during model construction each plugin can propagate information about possible program states, expressed as formulae of the common logic $\mathcal{L}$; currently a round of propagation happens at every successor computation. Our choice of $\mathcal{L}$ for this purpose is discussed in [5].

To make it easier to create a new model, the program and configuration details can be copied from an existing model and then modified; alternatively HECTOR generates reasonable default configuration options.

2.2 Drawing Abstract Models

HECTOR can draw the generated models in graphical form[1], as in Fig. 1. Each abstract state is drawn as a box containing a list of the formulae propagated between plugins during successor computation, and an illustration of its component for each plugin: (left-to-right) for constant propagation a partial map from variables to integers, for shape analysis a three-valued heap graph, and for trivector predicate abstraction a collection of (here four) abstraction predicates or their negations. CFG nodes are also shown, each one being "boxed in" with the set of abstract states collected there. The user can ask for all propagated formulae to be shown, or (as in Fig. 1) just those that turn out to affect the analysis. The statement shown in Fig. 1 appends a new node to a linked list, currently of length one. The shape analysis plugin shares a fact about reachability of nodes from the list head, enabling predicate abstraction to maintain the constraint that reachable list nodes have nonnegative data values.

[1] The Graphviz toolkit [7] is used for graph layout.

2.3 Ad-Hoc Model Checking

Once a model has been built, safety properties can be checked against it. Our safety language is "two-level": we take the *syntactic safety* LTL fragment from Thm. 3.1 in [10], but allow the "propositions" to be arbitrary $\mathcal{L}$ constraints on the program's current and initial states. This language can express quite a lot: for example,

$$\mathsf{G}\,(\mathit{allocd}(\mathit{node}, x) \rightarrow \mathsf{G}\,(\exists Y : \mathit{allocd}(\mathit{node}, Y) \wedge \mathit{next}(Y) = x))$$

says that, if the variable x ever becomes a reference to a linked list node, then forever after, x is pointed to by the *next* field of some list node (but not necessarily the same one in each future state). For convenience, however, shortcuts are available in Hector's interface for commonly used idioms, such as the absence of memory errors and assertion violations.

Model checking is performed using automata (generated by invoking scheck [8]), where we reuse Hector's machinery of sharing and successor computation to look up the values of the $\mathcal{L}$ "propositions" in abstract states.

If a counterexample is found it is shown to the user. If no counterexample is found, then the property holds of the original program. However (as usual after abstraction) counterexamples may be spurious, in which case a refined model with a better configuration may suffice to verify the property. Optionally Hector can search for what we call a *strong* counterexample, one in which the evaluation of the $\mathcal{L}$ "propositions" of the safety property yielded a definite answer in every state (as opposed to the third value "unknown"). While this still does not guarantee that the counterexample is feasible (because the existence of transitions is still uncertain), we conjecture that a strong counterexample will typically be more informative to the user than a weak one, even if the latter is shorter.

Also, similarly to what was observed in [3], a naïve model checking algorithm will lose precision if "propositions" have dependencies; our algorithm sometimes delivers a definite answer in these cases. For an example of this see [6].

Future work may involve adding CEGAR (counterexample-guided abstraction refinement) features to Hector. We are particularly interested in the possibility of a generic CEGAR algorithm which would work with all plugins, perhaps along the lines of the ideas put forward in [13].

References

1. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN Model Checking and Software Verification. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C programs. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, Springer, Heidelberg (2001)
3. Bruns, G., Godefroid, P.: Generalized model checking: Reasoning about partial state spaces. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 168–182. Springer, Heidelberg (2000)

4. Charlton, N.: Verification of Java Programs with Interacting Analysis Plugins. ENTCS 145, 131–150, Elsevier/Science Direct (2006)
5. Charlton, N.: Program Verification with Interacting Analysis Plugins. In: Formal Aspects of Computing, Springer, Heidelberg (2007), doi:10.1007/s00165-007-0029-4
6. Charlton, N.: HECTOR tool online, www.doc.ic.ac.uk/~nac103/hector/
7. GraphViz graph-drawing library: www.graphviz.org
8. Latvala, T.: scheck, www.tcs.hut.fi/~timo/scheck/
9. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM TOPLAS 24, 217–298 (2002)
10. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspects of Computing 6(5), 495–512 (1994)
11. Simplify theorem prover: research.compaq.com/SRC/esc/Simplify.html
12. TVLA, 3-valued logic analyzer: www.cs.tau.ac.il/~tvla/
13. Gulavani, B.S., Rajamani, S.K.: Counterexample Driven Refinement for Abstract Interpretation. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, Springer, Heidelberg (2006)