Exploiting Behavioral Hierarchy for Efficient Model Checking

Rajeev Alur, Michael McDougall, and Zijiang Yang

Department of Computer and Information Science
University of Pennsylvania

1 Introduction

Inspired by the success of model checking in hardware and protocol verification, model checking techniques for software have been the focus of a lot of research in the last few years [5, 3, 2, 6]. Model checking can be applied only to relatively small models due to its inherently high computational requirements, and there are two complementary trends to address scalability. The *model extraction* approach, exemplified by projects such as Bandera [6] and SLAM [3], involves constructing inputs to model checkers by abstracting programs written in languages such as C and Java. The *model-based design* approach, exemplified by modeling notations such as Statecharts [7], promotes design using high-level models that are compiled into code. Our research agenda is to develop model checking techniques for model-based design of software.

Modern software design languages promote *hierarchy* as one of the key constructs for structuring complex specifications. The input language to our model checker is based on *hierarchic reactive modules* [1]. This choice was motivated by the fact that, unlike STATECHARTS and other languages, in hierarchic reactive modules, the notion of hierarchy is *semantic* with an observational trace-based semantics and a notion of refinement with assume-guarantee rules. The first contribution of this paper is the *Hermes* toolkit that implements hierarchic reactive modules. Our implementation has a *visual* front-end and XML-based back-end, consistent with modern software design tools, and is in Java.

There are two basic techniques for reachability analysis. Enumerative model checkers such as SPIN [8] perform an on-the-fly exploration of the state-space using a depth-first search, while symbolic model checkers such as SMV [9] perform a breadth-first search by manipulating sets of states, rather than individual states, encoded typically by ordered binary (or multi-valued) decision diagrams. Since the two approaches are incomparable, and have been shown to be successful, *Hermes* supports both enumerative and symbolic reachability analysis. In this paper, we report progress on exploiting the structuring information in the behavioral hierarchy of the input model to speed up the the exploration of reachable state-space of the model for both the approaches. More information about the tool is available at http://www.cis.upenn.edu/sdrl/hermes/

D. Brinksma and K. G. Larsen (Eds.): CAV 2002, LNCS 2404, pp. 338-342, 2002.

2 Hierarchical Modeling in Hermes

Hierarchical Reactive Modules (HRM) is a graphical language for describing and analyzing systems. Our goal in using HRM is to find verification algorithms that leverage the modularity that is present in so many modern designs.

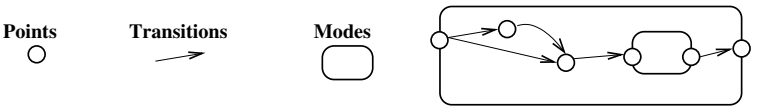

Fig. 1. The building blocks of the HRM language and a simple Mode diagram.

A simple HRM diagram resembles a finite state machine (FSM); it consists of states, called *points* in HRM, and transitions between points. HRM extends FSM by adding variables which can be read and updated as in normal programming languages. Each transition is enabled when its *guard*, a boolean expression over the diagram's variables, evaluates to true. Transitions can be annotated with *actions* which update the values of variables.

A set of points and transitions can be grouped into a *mode*. A mode's interaction with its surroundings is mediated by two interfaces: a *control* interface and a *data* interface. The control interface is a set of entry and exit points on the boundary of a mode. A mode can be embedded in other modes.

The data interface determines which data is available to a mode through a set of *global* variables. Each global variable is designated as *readable*, *writable* or both. Any external data that is not contained in a global variable will be hidden from the mode. Modes can also have their own *local* variables which are visible only to transitions within that mode or its submodes. A designer can re-use a mode by creating a *reference mode* which instantiates a copy of a mode that is defined elsewhere. Figure 1 shows a mode that contains some border points, internal points and one submode.

Certain modes are designated as *top level modes* that behave as separate processes or threads. We model concurrency by interleaving so that at any one time there is only one active top level mode. Top level modes communicate with each other through shared variables. HRM has a well-defined formal semantics [1].

HRM has two mechanisms for describing the requirements of a system. A point or mode can be given an *assertion condition*, which must be true whenever that point or mode is active. A system can also be given an *invariant*, which must be satisfied for all states of the system.

We have implemented a toolkit, called Hermes, which allows users to create, edit, type-check and verify HRM diagrams. The toolkit is implemented in Java and has a graphical user interface (GUI) for editing HRM diagrams. The GUI also acts a front-end to the model checking algorithms. Hermes also has command-line and scripting front-ends for environments where a GUI is impractical. The Hermes toolkit uses an XML file format to store HRM diagrams.

3 Enumerative Checker

The enumerative checker performs a depth first search of all reachable states of an HRM diagram. The search will check for states that are deadlocked or that violate the specified assertions or invariants. When the checker finds a bad state it outputs the sequence of steps that led to the bad state. The enumerative checker uses the structure and hierarchy of an HRM diagram to save time and memory while exploring the state space. Some of these techniques were described in [2]. In this paper we discuss some further optimizations.

Exploiting Hiding. A mode must declare which external variables it can read and write. The enumerative checker can use this information to create abstract views of a mode's context. Two contexts are considered equivalent if all the readable variables have the same value; unreadable variables may differ but this will not affect the behavior of the mode. If we have executed a mode in one context then there is no need to analyze the mode in subsequent equivalent contexts. Local variables are only visible to transitions within a mode so a top level mode will not be directly affected by the local variables of other top level modes. For example, suppose we have two top level modes: mode M with a local variable x, and mode N. If we have explored N's behavior in some context c_1 where $x = 3$, and we see another context c_2 which is the same as c_1 except that $x = 6$, then we do not need to compute N's behavior in context c_2. To exploit this, the enumerative checker keeps track of how each top level mode will behave in a context. When that context is seen again the old behavior is projected onto the current context to find the next reachable state.

Exploiting Sharing. This optimization exploits the fact that a mode can be shared by various parent modes. Recall that one mode may be instantiated in many places in an HRM diagram. Each instantiation will exhibit the same behavior when its global variables are the same. For example, Figure 2 shows a

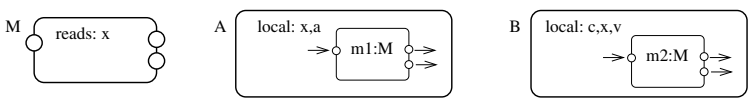

Fig. 2. Exploiting Sharing

mode M that is instantiated in two other modes A and B. Mode M only reads one variable x. The behavior of M will only depend on the value of x when its entry point becomes active. Once we have explored M in A with $x = 2$, we note which of the two exit points results from $x = 2$. When we encounter M in B with $x = 2$ we can just jump straight to the appropriate exit point. By suitable book-keeping, checker avoids recomputing a mode's behavior if another instance of that mode has already been searched for an equivalent context.

4 Symbolic Checker

Figure 3 (a) shows a mode M_1 with two submodes M_2 and M_3. M_1 has a local variable x, M_2 has local variables y_1, y_2 and M_3 has y_3, y_4. There are 8 transitions $t_1..t_8$ and 8 control points $c_1..c_8$. Given a typical symbolic model checker, the

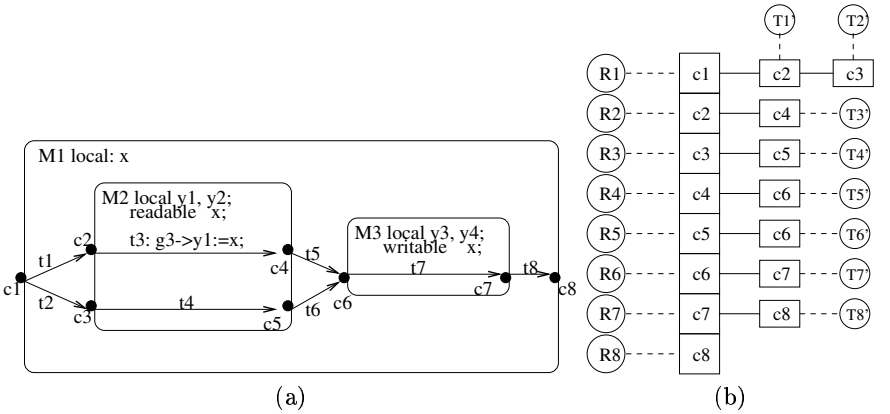

Fig. 3. (a): A hierarchical design. (b): Hierarchical symbolic representation

transition t_3 is denoted by an MDD representing $T_3 = (h = c_2 \wedge g_3 \wedge h' = c_4 \wedge y_1' = x \wedge x' = x \wedge y_2' = y_2 \wedge y_3' = y_3 \wedge y_4' = y_4)$, where the variable h is used to encode the control location. The transition relation of M_1, $T = \wedge_{i=1}^{8} T_i$, is represented by a single MDD or in conjunctively partitioned format.

In Hermes, the transition relation is represented as a map from control points to a list of pairs containing destinations of edges along with MDDs encoding guarded commands. Figure 3 (b) shows such a map for mode M_1. For each transition t_k from control point c_i to control point c_j, we build the MDD T_k' that encodes the guarded command of t_k. Then the pair (c_j, T_k') is added to the list associated with the control point c_i. Note that each T_k' is much smaller than the counterpart T_k used in a flat representation. For example, $t_3 = (c_2, c_4)$ can be denoted by a smaller MDD $T_3' = (g_3 \wedge y_1' = x \wedge y_2' = y_2)$. This is possible because local variables of M_2 and M_3 are not simultaneously active, therefore, y_3 and y_4 never appear in T_3' or other MDDs representing transitions in M_2. Since x is not writable in M_2, the term $x' = x$ can be dropped. The variables h is not needed in T_3' because the mapping provides information on control points. In other words, typing and scoping information of the original model is maintained during compilation of the transition relation using MDDs.

Like transition relations, the reachable state-sets in Hermes are not represented by a single MDD. A state region represented by an MDD is associated with each control point. As shown in figure 3 (b), there is an MDD R_i associated with each control point c_i. Such a representation allows us to partition the state

space intuitively with each region containing all the states with the same control point.

The reachability computation in Hermes computes reachable states at each control point. When a top mode M_i gets control for the first time, it starts the image computation from its entry point by following the transitions until the control gets stuck. The image computation returns an MDD S_i that contains the information about where and how the control inside M_i gets stuck. After each top mode has been given a chance to do the first image computation, it starts the next iteration by building a current onion ring for top mode M_i based on the stuck sets $\cup_i S_i$. The current onion ring is a map from the control points where the control became stuck during last image computation at M_i to newly reached states obtained from image computations at top modes other than M_i. By applying the image computation to the current onion ring of M_i, the control may continue from those stuck control points. The algorithm terminates if all the onion rings for top modes are empty, i.e., no new states can be reached at any control point.

In order to make Hermes work on existing sequential circuits designs we translate sequential circuits in BLIF format to XML that can be parsed by Hermes. The translation produces a Hermes model consisting of a single top-level mode. Besides having no concurrency, the model is linear rather than a tree or a DAG. Thus, the main structural feature that current Hermes exploits is the scoping of variables allowing for early quantification.

Acknowledgments. We thank Radu Grosu for helpful discussions. This research was supported in part by NSF award CCR99-70925, SRC award 99-TJ-688, and NSF CAREER award CCR97-34115.

References

1. R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proc. 27th POPL*, pages 390–402, 2000.
2. R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchical reactive machines. In *Proc. 12th CAV*, LNCS 1855, pages 280–295, 2000.
3. T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. 13th CAV*, 2001.
4. R. Brayton, G. Hachtel, A. Sangiovanni-Vincentell, F. Somenzi, et. al. VIS: A system for verification and synthesis. In *Proc. 8th CAV*, LNCS 1102, pages 428–432, 1996.
5. W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Trans. on Software Engg.*, 24(7):498–519, 1998.
6. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd ICSE*, pages 439–448. 2000.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
8. G. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engg.*, 23(5):279–295, 1997.
9. K. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.