The Murφ Verification System

David L. Dill

Computer Systems Laboratory
Stanford University
Email: dill@cs.stanford.edu

Abstract. This is a brief overview of the Murφ verification system.

The Murφ description language

Murφ is both a description language and a verifier for finite state concurrent systems [DDHY92]. It is appropriate for protocols and finite-state systems which can reasonably be modelled as a collection of processes that run at arbitrary speeds, where the steps of the processes interleave (only one process takes a step at any time), and where the processes interact by reading and writing shared variables. The Murφ verifier works by explicitly generating states and storing them in a hash table. We have put some effort into developing state reduction techniques, including symmetry reduction [ID93a, ID93b] , exploitation of reversible rules [ID96a], and verification of systems with varying numbers of replicated components [ID96b]. We have also investigated probabilistic verification techniques in Murφ [SD95c].

The Murφ description language was inspired by Misra and Chandy's Unity formalism [CM88]. A Murφ description consists of a collection of declarations of constants, data types such as subranges, records, and arrays, global variables, transition rules (which are guarded commands), start rules, and invariants.

The rules are similar to compound statements Pascal or Modula. Indeed, a rule can be arbitrarily complex, yet it is still executed atomically, meaning that the other rules cannot interfere. A *state* consists of the current values of the global variables. An *execution* of a Murφ program is any sequence of states that can be generated by starting in one of the states generated by a start rule, then repeatedly selecting a rule and executing it. Executing a rule generally changes the state, because the rule assigns to the global variables. Murφ is nondeterministic: there can be many executions, varying according to which rule was selected at each step of the execution.

A user can encode one of several concurrent processes by declaring variables for the process state and providing rules to capture its behavior. The behavior of several processes can be simulated by forming the union of the state variables and rules into a single Murφ program. Rule selection then simulates scheduling choices (the process whose rule is chosen runs next) as well as nondeterministic choice within a process.

Verification

The basic Murφ verifier generates all of the reachable states systematically, using a standard search algorithm such as breadth-first search. The search uses two data structures: a set of states whose descendants must be explored, and a table of states which have been previously encountered. When the search generates a state that is already in the table, the search is cut off. The invariant, which is a predicate which

reads the state variables, is evaluated in each newly generated state. If the result is *false*, verification halts and an error message is generated. The same effect can be achieved by an execution of an **error** statement in a rule. Similarly, if a state has no successors other than itself, the verifier halts and reports an error. In either event, the verifier also prints an execution from a start state to the offending state, to help with debugging.

We believe that explicit state verifiers are still useful, even when there are highly efficient BDD-based verifiers. One reason is that they are more predictable – performance is more closely related to the number of states, so the behavior of the verifier is more stable than with clever symbolic representations. The other reason is that some protocols, notably the ones we were most interested in verifying, require great cleverness to attack with successfully with BDDs. A naive approach performs much worse than Murφ. It is necessary to use non-obvious representations of state, identify variables that are functions of other variables, and/or decompose BDDs in various ways [HD93b, HD93a, HYD94]. Thus, verifying a such protocol with BDDs requires more expert users than attacking the same protocol with Murφ.

The basic Murφ verifier has been applied very successfully to several problems. It is especially suitable for multiprocessor cache coherence problems, because those were the problems we were working on most intensively when we were designing and redesigning the verifier. However, it has also been used for link-level protocols, a hybrid byzantine agreement algorithm, mutual exclusion algorithms, memory model specifications, and probably numerous other examples.

Symmetry reduction

In the last few years, we have found several ways of improving the performance of Murφ. The first was to exploit *symmetry* [ID93a, ID93b]. In some cases (particularly high-level descriptions of multiprocessor cache coherence protocols), components or values of a type can be exchanged arbitrarily without affecting the future behavior of the protocol. We have exploited this in Murφ by adding a new data type, called a *ScalarSet*, which is a subrange type with the additional restriction that it cannot be used in any way that "breaks the symmetry" between elements of the type (for example, there are no literal constants of the type, and one value cannot be compared with another using $<$). The Murφ semantic analyzer enforces these constraints, so that symmetry cannot be broken in the description.

Symmetry is exploited in the verifier by doing *symmetry reduction*. A *canonicalization function* is constructed by the verifier, which maps all states which are equivalent up to rearrangement of the elements of a scalarset to a particular representative state (a simple example of normalization would be sorting an array whose index set is a scalarset, if there are no scalarsets in the array itself). States are canonicalized before they are looked up or stored in the state table, so a state is not inspected if it is equivalent to a state in the state table, even if the states are not identical. This optimization has resulted in 100-fold reductions in the numbers of states generated in some cache coherence protocols. In certain cases (when a scalarset is not used as an array index), systems with unbounded scalarsets can be verified. For example, this property can be used to verify cache coherence regardless of the number of data values, and, hence, the number of bits in each data value.

Recent improvements

More recently, we have found an optimization which avoids storing transient states in the state table. The optimization works by identifying rules that do not lose information

when they are executed. The verifier can execute the "backwards" to map normalize transient states by finding a unique non-transient progenitor state from which they evolved [ID96a].

Most recently, we have developed a way of verifying certain systems with arbitrary numbers of replicated components in Murφ [ID96b]. The replicated components are flagged by using a datatype *RepetitiveID*, which is similar to a scalarset type but even more restricted. The verifier exploits this by working in an abstract space, where every global state is mapped to an abstract state which keeps track of whether there are zero, one, or more than zero of the replicated components in each component state. This method can be used to show that cache coherence protocols work properly for any number of processors. The method can be combined with symmetry reduction and the method of the previous paragraph to yield truly massive reductions in the state explosion problem.

We have also been exploring probabilistic verification algorithms, originally based on ideas from Gerard Holzmann, Pierre Wolper, and Denis Leroy [Hol87, WL, WL93], in which a small signature for each state is entered into the hash table instead of the state itself, saving a great deal of space at the expensive of some probability of producing a false positive result. The key is to find a bound on this probability, as Leroy and Wolper did. We have found several ways to reduce this bound, by changing the search and hashing algorithms and doing a more refined analysis of the probability [SD95a, SD95b, SD96]. This work has culminated in a factor-of-four reduction in the number of bits required per state, compared with Wolper and Leroy's original result, while guaranteeing the same or lower probability of missing an error,

Liveness

A few years ago, we implemented a version of Murφ which could verify common forms of liveness properties, expressed in a subset of linear temporal logic, using quite efficient state exploration algorithms. However, we have not updated the liveness verifier to use symmetry reduction and subsequent optimizations.

The Murφ verifier is available free by anonymous ftp from `snooze.stanford.edu` (directory /pub/murphi), under very liberal licensing terms.

Acknowledgements

The Murφ system was designed, implemented, redesigned, reimplemented, etc. by many different people, including me and the following students: C. Han Yang, Alan J. Hu, Andreas Drexler, Ralph Melton, C. Norris Ip, Seungjoon Park, and Ulrich Stern.

References

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design — a Foundation.* Addison-Wesley, 1988.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[HD93a] Alan J. Hu and David L. Dill. Conjunctive partitioned invariants for efficient verification with bdds. *5th International Conference on Computer-Aided Verification*, June 1993.

[HD93b] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th Design Automation Conference*, pages 266–271, 1993. Dallas, Texas, June 14–18.

[Hol87] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Protocol Specification, Testing, and Verification. 7th International Conference*, pages 339–344, 1987.

[HYD94] Alan J. Hu, Gary York, and David L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st Design Automation Conference*, pages 276–282, 1994.

[ID93a] C. Norris Ip and David L. Dill. Better verification through symmetry. *11th International Symposium on Computer Hardware Description Languages and Their Applications*, pages 87–100, April 1993. Extended version with complete proofs and semantic analysis to appear in Formal Methods in System Design.

[ID93b] C. Norris Ip and David L. Dill. Efficient verification of symmetric concurrent systems. *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, October 1993.

[ID96a] C. Norris Ip and David L. Dill. State reduction using reversible rules. *33rd Design Automation Conference*, June 1996.

[ID96b] C. Norris Ip and David L. Dill. Verifying systems with replicated components in murφ. In *this proceedings*, 1996.

[SD95a] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[SD95b] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224, 1995.

[SD95c] Ulrich Stern and David L. Dill. Improved probabilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[SD96] U. Stern and D. L. Dill. A new scheme for memory-efficient probabilistic verification. Submitted for publication, 1996.

[WL] P. Wolper and D. Leroy. Reliable hashing without collision detection. Unpublished revised version of [WL93].

[WL93] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification. 5th International Conference*, pages 59–70, 1993.