A Retrospective on Murφ

David L. Dill

dill@stanford.edu
Stanford University

Abstract. Murφ is a formal verification system for finite-state concurrent systems developed as a research project at Stanford University. It has been widely used for many protocols especially for multiprocessor cache coherence protocols and cryptographic protocols. This paper reviews the history of Murφ, some of results that of the project, and lessons learned.

1 Introduction

Murφ (pronounced "Murphy") is a formal verification system for finite-state concurrent systems. It was developed by my group in the Stanford University Computer Systems Laboratory during the 1990's. It has been a fairly successful project, resulting in a widely-used tool and in a number of published research results that have influenced other systems. Based on my discussions with members of design teams, it seems that Murφ has been used at some point in the development of the cache coherence protocols for almost every major commercial shared memory multiprocessor system. Murφ has also been used for a variety of other problems, including verification of cryptographic protocols [1, 2].

I learned much from the Murφ project about formal verification, the design of practical tools in an academic setting, and research strategy. This paper is a first-person recollection of how the Murφ project unfolded. It is intended to provide an overview of some of the issues and technical advances in explicit state model checking, a realistic portrayal of how a research project actually happened (information that does not appear in the technical papers about Murφ), and as a source of some hard-earned knowledge about building a formal verification tool in an academic environment.

2 Early years

2.1 The motivation for the project

The Murφ project came out of a desire to show that formal verification tools could have practical value. When I arrived at Stanford, in 1987, several faculty and many students in my building were involved in the DASH project [3], a large effort to design and build a shared-memory multiprocessor. The heart of the DASH system was a cache coherence protocol to maintain consistency between multiple distributed cached copies of data from memory. Not surprisingly, when I polled these colleagues about important problems in the hardware verification area, they frequently mentioned the challenge of assuring the correctness of cache coherence protocols.

At about the same time, my research group was exploring the use of Boolean decision diagrams (BDDs) for symbolic model checking of large state spaces [4]. A BDD is a data structure that compactly represents Boolean functions that arise in practice [5]. The essence of BDD-based model checking is to compute the reachable state space using only operations on BDDs, so that explicit lists or tables of states never have to be saved. BDD verification connected with cache coherence when I saw a talk by Ken McMillan, then a PhD student in Computer Science at Carnegie Mellon University, on verification of the Gigamax cache coherence protocol using BDDs [6].

A group of three PhD students, Andreas J. Drexler and Alan J. Hu and C. Han Yang and I decided to try the same techniques on a different cache coherence protocol. Ken had used the "m4" macro processing language to improvise a description language that could be translated into BDDs. We attempted a similar approach, but, after a few weeks, the students expressed the unanimous view that the method was far too labor-intensive, and that we could accelerate the project by first building a translator for a more user-friendly description language. That language eventually became Murφ.

2.2 The Murφ description language design

The basic concepts of the Murφ description language sprang from the UNITY modeling language of Misra and Chandy [7], which I had learned about a few years earlier, and found to be simple and appealing. In the UNITY model, a concurrent system is represented as a set of global variables and guarded commands. Each guarded command consists of predicate on the state variables, called a *guard*, and a set of assignments that update those variables to change the state. The "control structure" of a UNITY program consists of a single infinite loop, which repeatedly executes two steps: (1) evaluate all the guards, given the current values of the global variables and (2) arbitrarily choose one of the commands whose guard is true and execute it, updating the variables.

A UNITY model defines an implicit state graph, where each state is an assignment of values to the global variables. The initial values of the variables are specified as part of the model. The commands define the next-state relationship: if a command has a guard that is true in state s, and state t is obtained by executing the assignments of that command, then there is an edge from s to t in the global state graph.

Modeling concurrency in UNITY is trivial. UNITY has no notion of a process or thread, but a process can be represented as a set of variables and commands. The joint behavior of several processes is simply the union of the variables and commands for each process. This approach provides an asynchronous, interleaving model of concurrency, where all synchronization and communication is through global variables. Nondeterministic choices by a scheduler are subsumed by the nondeterministic choice of which guarded command to execute on each iteration of UNITY's loop.

While preserving the basic concepts of UNITY, we quickly found that it was very helpful to have more conventional programming language constructs for the basic data types of the language (records, arrays, integer subranges, enumerated types, etc.) and for the predicates and statements appearing in the commands. We even included "while" loops in the commands, in spite of some initial concerns about potential problems from non-terminating commands (in retrospect, these concerns were completely unimportant – infinite loops in commands are are rare, and easily detected and fixed). Pointers and

heap-allocated memory were not included because of a desire to discourage the writing of descriptions requiring very large state representations. It is not clear to me that this was a good decision, since it resulted in significant inconvenience when modeling some software applications.

Murφ could check several kinds of properties. It could detect deadlocks, which were defined as a state with no successors other than (possibly) itself. Murφ descriptions are supposed to be deadlock-free. The user could also specify *invariants*, which are predicates on the state variables (in the Murφ language) that are supposed to hold in all global states. We soon discovered that assertions and error statements embedded in commands also very useful, because it is often easier to specify an error condition in the context of the logic of a rule than to separate it from the rule. For example, users frequently handle a range of possibilities with a "case" statement or nested conditionals, and want to add "otherwise, it is an error" for all the cases that are not supposed to occur. Whenever a new state is constructed, these properties are all checked and an error message and error trace are produced if one is violated.

The Murφ model could be translated to logic by writing each command as a logical relation, and then representing the next-state relation as a disjunction of predicates, one for each Murφ command. This translation seemed attractive for BDD-based verification, because BDD-based evaluation of reachability has an outermost existentially quantified variable which can be distributed over the disjunction of predicates, minimizing the size of intermediate BDDs generated during the computation (this trick is very important to BDD computations; it is called "early quantification").

The three students began implementing the system. My recollection is that Andreas worked on the description language translation, Alan worked on the BDD verifier, and Han worked on examples. Unfortunately, we soon ran into a barrier: There were efficiency problems with the BDD verifier. As a interim measure to allow Han to work on examples before the BDD verifier was functioning we decided to implement a simple explicit-state on-the-fly verifier for Murφ, with the intention of throwing it away in a few months.

Explicit on-the-fly verification is a simple depth-first or breadth-first search of the state graph for a state that violates a property. An on-the-fly algorithm checks each state as it is created, so that an error can be reported before the entire set of reachable states is explored. Such methods had already been used for protocol verification for over a decade. The search algorithm uses two large data structures: a queue of states whose successor states need to be searched (the *queue*) and a hash table of states that do not need to be visited again (the *state table*).

2.3 Murφ's first major application

Shortly after we had our first working prototype of Murφ, I connected with Andreas Nowatzyk, who was designing an experimental shared-memory multiprocessor at Sun Microsystems called S3.mp [8]. I knew Andreas from the PhD program at Carnegie Mellon University, where he had shared an office with Michael Browne, another student of Ed Clarke's who was working on model checking. Andreas became familiar with model checking through his officemate, and later worked with Ken McMillan, who found a bug in an early version of the cache coherence protocol that Andreas was

designing. We decided to apply Murφ to this problem. I, personally, did much of the work on this, and soon became the main user of Murφ. I generated a long list of requests for improvement in the language and the system to make it more usable. Murφ improved rapidly, and we found several bugs and other issues in the protocol. Later, Han Yang applied Murφ to many problems related to S3.mp.

When working on the S3.mp protocol, we learned some things about how to use Murφ. For example, we discovered the importance of *downscaling*. It was impossible to represent the protocol at the scale it was implemented, with potentially hundreds of processors and millions of cache lines. Instead, bugs could be often found with scaled-down models having, for example, three processors and one "cache line" with only one bit of data. Using such a model, the verifier could exercise many subtle scenarios and find bugs quickly in such a model that would be very difficult to find with system-level simulation. Also, when a problem with a scaled-down model was discovered, it was much easier to understand the problem than it would have been using a larger-scale model [9]. Indeed, it was a good idea to try the smallest possible model that made any sense at all, find and fix any bugs, and then scale the model up slightly and repeat the process. Changing scales was also useful when Murφ ran up against capacity limits, by allowing the user to increase the scale of one dimension (*e.g.*,the number of cache lines) while decreasing the scale of another dimension (*e.g.*, the number of processors).

It became obvious that Murφ needed to support verification of the same model at different scales with minimal changes to the description. We found that easy rescalability could be achieved with three simple features: (1) named constants, (2) arrays, and (3) parameterized sets of commands. For example, a generic model could be written for n processors. Each processor could have an associated numerical index in the range $1 \ldots n$, the global state variables for processor i could be stored in one or more arrays at index i, and the guarded commands for the processor could be nested inside a *ruleset*, a new Murφ construct that defines a symbolic parameter to represent a process index. Each command inside a ruleset was actually a *family* of commands, one for each possible value of the ruleset parameter. Each command could use the process index to access the appropriate array element, and as a storeable value for the name of the process (for example, the process index might be stored in a variable representing the owner of a cache line). With such a representation, the number of processors verified could be changed simply by re-defining the symbolic constant n.

After these improvements, we found that useful results could be obtained with our "interim" explicit-state implementation, and that we could compensate for some of the obvious efficiency problems with the verifier by judicious use of abstractions, as well as generally minimizing the number of state variables.

2.4 BDD research

Meanwhile, Alan Hu was still struggling with the BDD-based verifier for Murφ. He defined a somewhat simpler input language (eventually called "EVER"), so that he could find out if BDDs could be made to work efficiently for the problems of interest before investing the effort to handle the entire Murφ description language. It seemed that our distributed cache implementations led to BDD blowup regardless of what we did. It eventually dawned on us that there was an inherent problem: Many of the transactions

in the cache model involved sending a message from processor A to processor B and storing the message. In the midst of such a transaction, processor A would be in a state waiting for a response to the message it sent, processor B would be in a state waiting for a message, and the message itself would have "from" address of A and a "to" address of B. All of these relationships would be implicit in a BDD representing the reachable states of the system.

BDD variables must be placed in a single total order. As a rule, BDDs explode in size when highly correlated variables are widely separated in this total order. But there seemed to be no way to avoid separating closely correlated variables in the system we were modeling, because there could be many transactions in process at the same time, and transactions could involve any pair of processors. So, a variable order that put the variables for processor A and B near each other would separate variables for A and C, A and D, etc., which also correlated. So the BDDs would always blow up.

We came up with a two ideas for reducing this problem. The first was to note that, in many cases, when there were several related variables, some of them were actually redundant – their values could be inferred from the values of other variables. For example, the state of a process that has just sent a message might be a determinable from the contents of the message. One idea was to declare variables that are *functionally dependent* on other variables, and treat them as abbreviations rather than as separate variables [10]. This reduces some of the redundancy that causing BDD blowup. Since the declarations of functional dependencies could be wrong, the verification algorithm verifies that there really is a functional dependency before exploiting it.

Another, more general, idea was to maintain a list of separate BDDs for sets of related variables, such as those in an individual processor or in the network, the conjunction of which represented the reachable state space. Most BDDs would capture the relationships between a small set of related variables, such as the variables describing the state of a processor. Since these BDDs had a small number of variables, the related variables could be close together. Other BDDs could focus on different properties involve different sets of variables (the sets of variables did not have to be disjoint). There would be a BDD at the end of the list that included all the variables, but the relations captured in the earlier, smaller BDDs in the list would be factored out of the large one, to keep it small. So, in the best case, we would have several small BDDs instead of one huge BDD [11, 12].

Both of these ideas worked moderately well on artificial examples, but we still could not handle the S3.mp cache coherence protocol, and we eventually concluded that there were several different causes of blowup, each of which was going to require a complicated solution – if all were even solvable. We were never able to verify distributed cache coherence algorithms with BDDs. To my knowledge, no one else has been able to use BDDs successfully for this problem, even now.

3 Optimizations

3.1 State Reduction

Subsequently, another student, C. Norris Ip, joined the project. Norris and I noticed that, in many of the examples, Murφ was searching many redundant states because of sym-

metry in the system description. For example, some of our cache coherence examples would have three different kinds of symmetry. Processors, cache lines, and memory locations could all be interchanged in a state without changing the future behavior of the system in any important way. More precisely, once a state had been visited and its descendents searched without finding a bug, it was pointless to search a state that was identical except for a permutation of the processors [13, 14].

We also noticed that symmetries in our descriptions corresponded to the use of numerical indices, which were represented as subrange types in Murφ. Equivalent states corresponded to different permutations of the elements of these subranges. If Murφ could standardize each state to find an equivalent representative state before looking it up in the state table, the search algorithm would only search the descendants of the representative state instead of all the states equivalent to it. It turned out that standardizing states was a difficult problem (at least as hard as graph isomorphism), but it was possible to find reasonably accurate and fast heuristics.

However, if symmetry reduction were applied inappropriately, identifying states that were not equivalent, Murφ could miss errors. The solution was to invent a new, more abstract type for subranges (which we called a "scalarset") values of which could only be assigned to variables, used as array indices, or used as ruleset parameters. Under these restrictions, it could be guaranteed that permuting the elements of a scalarset (which were represented as small integers) would preserve state equivalence. So, symmetry could be "declared" by change subrange declarations to scalarsets, and the Murφ compiler would detect any symmetry-breaking operations that might lead to missed error states.

We made an important discovery when looking at graphs of the numbers of states searched as a function of scalarset sizes. For some scalarsets, the number of states stopped increasing even as the size of the scalarset was increased. We realized that Murφ could automatically verify an infinite family of systems, completely automatically, under certain circumstances! For example, if the scalarset type s was only stored in three places in a state, the representative state would map the original values to 0, 1, and 2 – even if s had a hundred possible values. We called scalarsets that were used in this limited way "data scalarsets," and the phenomenon where the state graph stops growing "data saturation."

Scalar sets only capture *full symmetry*, in which the members of the set can be permuted arbitrarily. Other researchers have explored other types of symmetry that can occur in systems, [15–18] but full symmetry is easy to express, occurs frequently in practice, and saves more states than other types of symmetry, so we made a choice, in the engineering of Murφ and in our research strategy, not to pursue the topic more deeply.

3.2 Reducing the primary memory bottleneck

One of the major problems with explicit on-the-fly verification, as described above, is poor locality of reference in the state table. The queue needs to store entire states, because all of the information in each state may be necessary to compute the successor states, so it uses a lot of memory. But the queue has good locality of reference – states that are in use tend to be located in memory near other states that are in use – so most

of the states can be migrated to the disk without major losses in efficiency when they verifier is not using them. However, the state table, being a large hash table, has no locality of reference. Once the size of the state table approaches the size of available primary memory, parts of it are paged out to the disk, resulting in slowdowns of many orders of magnitude. Thus, available primary memory effectively limits the number of states than can be searched.

Ulrich Stern, then a visiting student from Germany, joined our group and began investigating whether we could reduce memory usage by using probabilistic techniques such as the bit-state hashing method pioneered by Holzmann [19, 20] or the hash compaction method that had recently been published by Wolper and Leroy [21, 22]. These techniques save memory, but at the expense of some probability that states (and, possibly, errors) will be missed. Bit-state hashing does not provide any guarantees about that probability, but Wolper and Leroy could provide an upper bound on the probability of missing an error, if the verifier completed without finding any errors.

Instead of storing full states in the state table, the Wolper/Leroy scheme stored numerical signatures, computed using a hash function. The signatures were much smaller than the states. In this method, states could be missed when they have the same signature. If the signature of state s_1 is stored in the table, and s_2 with the same signature is looked up, the verifier will find the signature of s_2 in the state table and mistakenly conclude that the descendents of s_2 have already been searched, and never visit them. If all of the error states are descendents of s_2, this can result in missed errors.

Wolper and Leroy could bound the probability of missed errors after the completion of the search by counting the number of states actually visited and computing the probability that two of them had the same signature. This number could be made reasonably small if there were sufficiently many bits in the signatures. In theory, the user could raise that small probability to the n^{th} power by verifying n times with different hash functions.

Uli's first improvement to this scheme was to improve the probability bound by noticing that a state could only be missed if it had the same signature as another state *and* hashed to the same location in the hash table – so the log of the number of buckets in the hash table could be used in place of bits in the signatures.

A more significant improvement came from using breadth-first search. A particular error state, e, can be missed only if some state on a path from the start state to e is missed due to a signature collision. Hence, the probability of missing e can be minimized by minimizing the lengths of the paths searched by the verifier – which is what breadth-first search does. If search completes with no errors, the probability that this answer is incorrect is no greater than the probability that state e was missed. That probability can be computed, after verification, from the maximum number of breadth-first search levels and the number of states in the hash table. Empirically, many of the state graphs of our applications had small number of levels, so this improvement resulted in another 50% reduction in the number of bits required per state, while preserving the same small probability of missed errors.

Uli also discovered a way to make effective use of secondary storage for the state table. This idea was inspired by an earlier algorithm by A.W. Roscoe [23]. Uli discovered a simpler and more efficient scheme, which again relied on breadth-first search. Only

the current level of breadth-first search states are kept in primary memory. Redundant states within the same level are detected using a hash table in primary memory. After all the states in a level have been generated, the stored states on the disk are scanned linearly, and states are deleted from the newly-computed breadth-first layer when those states are read off the disk. In practice, this algorithm allowed orders of magnitude more states to be searched, with a small percentage increase in computational overhead.

Parallel Murφ

Hash compaction and the improved use of disk storage converted the storage bottleneck to a CPU-time bottleneck, in many cases. Fortunately, Uli also devised a search algorithm that made effective use of parallel computing to reduce the CPU bottleneck.

The basic algorithm is simple. The state table is partitioned among many processors, and each processor "owns" the states that hash to its part of the state table, so, essentially, the states are allocated randomly to processors. Newly created states are sent to the processors that own them. Those processors check whether the states they were sent are already in the state table. If not, the processor computes the successors of the state and sends those to their owners. Uli also devised an efficient (and correct) distributed termination detection algorithm that worked well in practice.

Parallel Murφ is surprisingly effective at balancing the load. It exhibited linear speedup and high efficiency even with relatively large numbers of processors (e.g., a speedup of a factor of 44 to 53 when running on a 63-processor system).

Conflict with liveness checking

The original plan for Murφ was to implement checking for liveness properties as well as simple safety properties and deadlock checking. Seungjoon Park, another Stanford PhD student, implemented checking for properties a simple subset of linear temporal logic (LTL) formulas that captured some of the most common liveness and fairness properties, using a modified depth-first search algorithm. However, it seemed that every major efficiency improvement conflicted with these algorithms. Symmetry reduction of a state graph, as described above, does not preserve liveness properties, and it is not simple to modify the symmetry reduction or liveness checking algorithms to make them compatible (Later, Gyuris, Sistla and Emerson implemented clever liveness checking algorithms that exploit symmetry reduction [18, 24]).

The state table and disk optimizations relied heavily on breadth-first search. But on-the-fly verification of liveness properties seem generally rely on depth-first search algorithms. Unfortunately, an efficient on-the-fly breadth-first search algorithm for liveness properties still does not exist.

Given a choice between verifying safety properties and liveness properties, safety properties are probably more important in practice. However, it would be better not to have to choose, because liveness properties are also extremely important. I have had several students in class projects develop protocols that they verify successfully in Murφ only because the protocols livelock before they can reach an error state.

4 Lessons Learned

Our team learned many lessons about verification, research strategy, and tool design from the Murφ project. Unfortunately, I have validated many of these lessons by ignoring them in other projects, and suffering the consequences. Findings such as these are rarely written in technical papers. I am writing them here in the hope that they may be of value to others.

Bug hunting is more rewarding than proving correctness. I had already learned this in my graduate work, but our work on Murφ strongly reinforced the lesson. Finding bugs in a system is usually much easier than proving correctness, and the impact is usually greater. Designers of systems implicitly believe that the systems are correct, so they are impressed, if not pleased, when a bug is found. Additionally, some people don't believe proofs of correctness, but a bug is explainable and demonstrable. This is the justification for using Murφ and similar tools on partial, abstracted, scaled-down models of systems.

Start using the tool to solve real problems as early as possible, and make whatever changes are necessary to maximize the tool's usefulness In the the best case, the user is an implementer of the system (or, even better, the manager of the project, as I was in the early days). The tool should be put to use as early as possible, so that improvements in the tool can be driven by the demands of the problem. Every PhD student who maintained (and rewrote) Murφ was also worked on challenging verification problems using Murφ.

This heuristic is helpful to ensure that a tool is useful for at least one thing. That may not seem to be an ambitious goal, but it is frighteningly easy to make tool that is useful for *no* applications, because of faulty intuition about the answers to crucial questions, such as "What are the critical language features?" and "Where are the performance bottlenecks?" However, using the tool on a real problem very rapidly reveals these misconceptions. Useful additional features and optimizations will become apparent with use; more importantly, many complex and difficult features will never be implemented because they are never really needed.

To keep the tool in use, it is necessary to respond to demands quickly. This pressure leads to creative enhancements to the tool that solve the user's problems with minimal redesign and implementation – and these solutions often turn out to be better for the system and user than a feature requiring more elaborate implementation. (However, over time, a certain number of sub-optimal but expedient decisions accumulate, and the tool needs to be redesigned and reimplemented more-or-less from scratch. Murφ has been completely re-written at least four times.)

Almost every decision in the design of the Murφ language and verifier was profoundly affected by this application-oriented philosophy. We abandoned BDDs because we found they did not work for the application, and the explicit state verifier evolved from incremental improvements from an algorithm that we initially thought would be impractical. The invention of scalarsets came from examination of the symmetries in

the application, and from the pervasive use of small integer subranges to represent sets that turned out to be symmetric.

Make it easy for people to audition the tool. Potential users are impatient, and they have to encounter many broken and/or useless tools on the Internet. It is time-consuming and frustrating to sort through all the advertised systems that are available for a particular process to find out which ones actually work. My unscientific estimate is that the potential user community for a tool declines exponentially with the number of minutes it takes to see whether a tool works.

The tool should be easy to download and run. I'm amazed at the number of tools that require a user to download five different programs and libraries from different sources before anything will compile. Of course, some of those programs are broken, or conflict with other software on the user's system, etc. It is best to make pre-compiled binaries available, and statically link them (when possible) so that library version problems don't arise, and otherwise do whatever can be done to make sure the tool works "out of the box." When a tool doesn't work, sometimes highly motivated or obsessive users will fix it, if they have access to the source code – so it is good to distribute source code.

Murφ did well in some regards and not so well in others. We *did* distribute binaries for linux and some other Unix operating systems. We did not support Windows – although, at that time, most interested users were probably running on Unix systems. The biggest problem with Murφ was that the compiler translated the Murφ description to a C++ program that then had to be compiled and executed to search the state space (this idea was borrowed from SPIN [25]). The problem with this approach is that C++ semantics would shift from version to version of the compiler. In particular, semantic analysis often became more stringent with new releases, so that generated C++ code would suddenly produce errors or warnings because a user tried to compile with a newer version of g++. A more pragmatic solution would have been to compile to a least-common-denominator dialect of C, but a series of implementers of Murφ found it much easier to generate C++ – so that's what they did.

Minimize intellectual property issues. We initially considered whether we should protect the potential commercial value of Murφ by restricting the license for noncommercial use. After conversations with potential users at some very large corporations, it eventually became clear that we, as well as potential users, would have to spend far too much time talking to lawyers before someone at these companies could use our tool. Even the Gnu Public License (GPL) created complications. We thought explicitly about our strategy, and concluded that the impact of our research would be maximized if we made the license terms as liberal as possible. We then adopted something very similar to the MIT and Berkeley licenses, which allow users great freedom in using, modifying, and incorporating code into other systems. I'm not sure Murφ would have been at all successful had we done otherwise.

Acknowledgments

Murφ was the work of many students, and occasional post-docs and staff, at Stanford. The roles of Andreas Drexler, Alan J. Hu, and C. Han Yang, Norris Ip, Seungjoon Park, and Ulrich Stern are partially described above. Andreas, Norris, and Uli each almost completely rewrote the system. Unfortunately, space does not permit discussion of many of the contributions of each, including large examples verified, variations of Murφ developed, and papers written.

In addition, Ralph Melton (an undergraduate at that time) maintained and almost completely rewrote the system, and Denis Leroy (a Master's student at the time) maintained and improved it. There have been so many helpful collaborators that I'm bound to forget some, but some of them were Satyaki Das, Steven German, Ganesh Gopalakrishnan, Richard Ho, Paul Loewenstein, John L. Mitchell, Andreas Nowatzyk, John Rushby, Jens Skakkebaek, and Vitaly Shmatikov.

The Murφ project was supported financially by a series of sponsors, including the National Science Foundation MIP-8858807, the Defense Advanced Research Projects Agency (through contact number N00039-91-C-0138, and, later, NASA grant NAG-2-891).), the Semiconductor Research Foundation (contract 95-DJ-389), financial gifts from the Powell Foundation, Mitsubishi Electronics Research Laboratories, the Stanford Center for Integrated Systems, equipment gifts from Sun Microsystems, Hewlett-Packard, IBM, and Intel.

The Murφ project resulted in a tool that has no doubt made widely-used systems notably more reliable. It also trained some of the most talented computer technologists in the country, who are now key contributors at companies and universities. I believe that the support of the above sponsors has yielded benefits to the U.S. and the world that are many times greater than the investment. I would like not only to thank the sponsoring organizations, but the individuals within them who were responsible for deciding to fund our project in the hope that we would produce something of value.

Of course, any statements, findings, conclusions, or recommendations above are my own, and do not necessarily reflect the views of any of the project funders.

References

1. Mitchell, J.C., Mitchell, M., Stern, U.: Automated analysis of cryptographic protocols using murϕ. In: IEEE Symposium on Security and Privacy. (1997) 141–153
2. Mitchell, J.C.: Finite-state analysis of security protocols. In: Computer Aided Verification. LNCS (1998) 71–76
3. Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.D., Gupta, A., Hennessy, J., Horowitz, M., Lam, M.: The Stanford DASH multiprocessor. Computer **25**(3) (1992)
4. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. 5th IEEE Symposium on Logic in Computer Science (1990)
5. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35(8)** (1986)
6. McMillan, K.L., Schwalbe, J.: Formal verification of the gigamax cache-consistency protocol. In: Proceedings of the International Symposium on Shared Memory Multiprocessing, Information Processing Society of Japan (1991) 242–251

7. Chandy, K.M., Misra, J.: Parallel Program Design — a Foundation. Addison-Wesley (1988)
8. Nowatzyk, A., Aybay, G., Browne, M., Kelly, E., Parkin, M., Radke, W., Vishin, S.: The s3.mp scalable shared memory multiprocessor. In: International Conference on Parallel Processing. (1995)
9. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. IEEE International Conference on Computer Design: VLSI in Computers and Processors (1992) 522–525
10. Hu, A.J., Dill, D.L.: Reducing BDD size by exploiting functional dependencies. 30th Design Automation Conference (1993) 266–271
11. Hu, A.J., Dill, D.L.: Efficient verification with BDDs using implicitly conjoined invariants. 5th International Conference on Computer-Aided Verification (1993)
12. Hu, A.J., York, G., Dill, D.L.: New techniques for efficient verification with implicitly conjoined bdds. 31th Design Automation Conference (1994) 276–282
13. Ip, C.N., Dill, D.L.: Better verification through symmetry. 11th International Symposium on Computer Hardware Description Languages and Their Applications (1993) 87–100
14. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design **9**(1/2) (1996) 41–75
15. Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design **9**(1/2) (1996) 77–104
16. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design **9**(1/2) (1996) 105–131
17. Emerson, E., Sistla, A.: Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. 7th International Conference on Computer-Aided Verification (1995)
18. Gyuris, V., Sistla, A.P.: On-the-fly model checking under fairness that exploits symmetry. Formal Methods in System Design **15**(3) (1999) 217–238
19. Holzmann, G.J.: On limits and possibilities of automated protocol analysis. In: Protocol Specification, Testing, and Verification. 7th International Conference. (1987) 339–44
20. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice-Hall (1991)
21. Wolper, P., Leroy, D.: Reliable hashing without collision detection. In: Computer Aided Verification. 5th International Conference. (1993) 59–70
22. Wolper, P., Leroy, D.: Reliable hashing without collision detection. (Unpublished revised version of [21])
23. Roscoe, A. In: Model-checking CSP. Prentice-Hall (1994)
24. Sistla, A.P., Gyuris, V., Emerson, E.A.: Smc: a symmetry-based model checker for verification of safety and liveness properties. ACM Transactions Software Engineering Methodolgy **9**(2) (2000) 133–166
25. Holzmann, G.: Design and validation of computer protocols. Prentice Hall (1991)