RESEARCH FEATURE

Practical Verification of Embedded Software

Using a new verification algorithm called the compositional backward technique, the authors demonstrate that they can exhaustively verify even the largest industrial applications—comprising more than 1,000 components—in a few minutes on a standard PC.

Jørgen Staunstrup
DHI

Henrik Reif Andersen

Henrik Hulgaard

Jørn Lind-Nielsen
IT University of Copenhagen

Kim G. Larsen

Gerd Behrmann

Kåre Kristoffersen

Arne Skou
Aalborg University

Henrik Leerberg
IAR Systems A/S

Niels Bo Theilgaard
Baan Nordic A/S

Advances in processor speed, memory capacities, sensors, and peripherals have enabled the inexpensive fabrication of sophisticated products ranging from mobile phones and hi-fi equipment to highly complex software in cars and airplanes. Unfortunately, the lack of good design methods and tools is a major bottleneck in the development of these products, particularly those with a short life cycle such as consumer electronics and household appliances. Developing embedded software for large, complicated applications requires models that are both intellectually manageable and physically realizable. Choosing a modeling technique is a compromise between conflicting goals: Models must be easy to comprehend and construct, but they also must be practicable and provide platforms for analysis.

Academia and commercial tool developers have proposed various embedded software models that represent different emphases on these goals.[1-3] In the model we describe, efficient realizations and correctness receive high priority at the expense of descriptive features.

Because embedded software is firmware—and therefore difficult or impossible to replace—its correctness is of paramount importance. Furthermore, embedded applications are often manufactured in large quantities, making it expensive to correct software errors. Exhaustive verification—a technique that implicitly checks all possible computations—is a practical alternative for ensuring the correctness of embedded software. Our work demonstrates that the visualState commercial design tool can verify even the largest industrial applications—comprising more than 1,000 concurrent components—in a few minutes on a standard PC.

The compositional backward technique is a new algorithm that dramatically improves runtimes compared with the algorithms traditionally used for exhaustive verification. We developed this algorithm to check safety properties, but it has been extended to handle a larger class of properties including liveness.[4-5]

Our algorithm dramatically improves verification runtimes by decoupling independent states and collapsing states that behave similarly. We have obtained encouraging results using this algorithm to exhaustively verify embedded controllers used in large industrial applications. Previous versions of visualState (http://www.visualstate.com), the commercial tool that incorporates the compositional backward technique for developing embedded software, have been used in hundreds of industrial applications.

EXHAUSTIVE VERIFICATION

As with most other software, manufacturers use tests to verify the correctness of embedded software. However, a simple example demonstrates why—despite its extensive use—software testing is grossly insufficient to ensure the correctness of even modestly complex embedded software. Assume that a human operator uses the control panel shown in Figure 1 to control two plane motors. The control panel has several buttons and two warning lights (the solid bars on top). Each button incorporates a small light showing whether the button is enabled—that is, whether pressing it will have some effect. From time to time, each subpanel enters a critical phase indicated by the warning light at the top, showing that the corresponding motor is becoming overheated. The plane can still operate if one subpanel is in the critical phase; however,

0018-9162/00/$10.00 © 2000 IEEE

both subpanels must never go critical simultaneously.

Testing the control panel by repeatedly pushing enabled buttons and checking that the double-critical phase never results would probably persuade a designer that it is operating correctly.[6] (You can try this yourself on an interactive demo of a random test at http://www.itu.dk/research/vvs/verian.html. So far, no one has found a manual sequence of button activations that results in both subpanels going critical.)

Although we have run simulation tests—activated by randomly pushing one of the two to seven enabled buttons on the control panel—for several weeks, we have not found a computation that causes both subpanels to go critical simultaneously. After trying the interactive version and watching the test at length, you might conclude that this is a working design in which it isn't possible to trigger the critical phase for both components simultaneously. However, you would be wrong. It is indeed possible to push enabled buttons in a sequence such that both functions enter their critical phases. This sequence (consisting of 38 steps) may be found in a few seconds using exhaustive verification.

Because this example is much simpler than many applications found in practical use, it is surprising that the bug that the exhaustive verification reveals is not found by testing the application on a set of sample data.

This example in which the operator chooses between two or three different operations in each step of the computation is typical of embedded applications. For example, mobile phone or remote control applications typically entail a choice between a small set of actions at any one point. If there are k choices in each step, after n steps there are k^n different test sequences. If only a small fraction of the k^n sequences leads to an error, conventional testing by simulation is highly unlikely to detect the error. In most cases, only a systematic generation of test data, which essentially amounts to an exhaustive verification, reveals such tricky mistakes.

Most of the spectacular examples of software bugs that have caused major damage or great loss resemble this example. In each case, an unexpected combination of events led to a state that was neither anticipated nor tested. The lesson in each case seems to be the same: Designers need techniques that allow them to exhaustively traverse the reachable state space.

During the past 10 to 15 years, designers have provided a number of data structures and algorithms capable of handling very large practical examples.[4, 7-9] Although theory tells us that there must be examples that cannot be handled, our experience indicates that these examples rarely occur in practice.

THE STATE MACHINE MODEL

We use a state machine model that describes a computation as transitions between a fixed set of states. The visualState tool is a conceptually simple state machine model that has received widespread practical use. The state machine model primarily captures the computation's control flow; it handles data manipulation such as arithmetic and nontrivial data structures separately. This distinction between control and data is common practice in hardware design, and it is also a very useful distinction in software design.

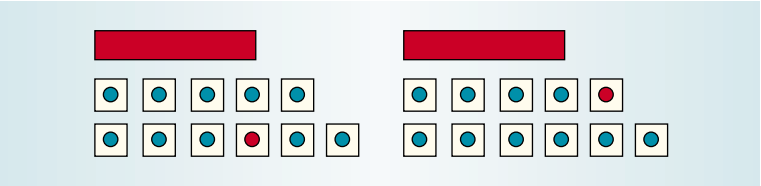

Figure 1. Control panel for controlling two plane motors. Each button has a small light showing whether it is enabled. The red warning light at the top shows whether the corresponding motor is becoming overheated.

The visualState tool is an extension of a Mealy machine that allows concurrency.[10] It can be viewed as a simplified version of Statecharts[2] and the Requirements State Machine Language.[3] The design consists of a fixed number of concurrent, finite-state machines that have pairs of input events and output actions associated with the transitions on the machines. The computational model is synchronous: All machines react upon each input event in lockstep; concatenating the output that the individual machines generate yields the total output. Adding transitions with guards expressing conditions on the local state of other machines leads to further synchronization between machines.

The simplified videocassette recorder model shown in Figure 2 comprises three state machines:

- MOTOR determines the tape's speed and direction. That is, the tape can either stop (STOP), move forward (FW), or move backward (BW).
- POWER indicates whether power is on (ON) or off (OFF).
- HEAD indicates whether the playing head is up (UP) or down (DOWN).

The machines change state as reactions to the input events: *on*, *off*, *up*, *down*, *stop*, *FW*, and *BW*. For all events except *stop*, at most one machine is reacting. The event *stop* synchronously resets the HEAD and the MOTOR to their initial states: *up* and *stop*. In this design, some transitions are guarded with conditions on local states of other machines. In particular, the HEAD can only be lowered once POWER is turned on. The MOTOR in turn can only be brought to move (forward or backward) once the HEAD is lowered. This simple design does not generate any actions or outputs. However, in a more complete design, transitions would generate actions for physically lowering

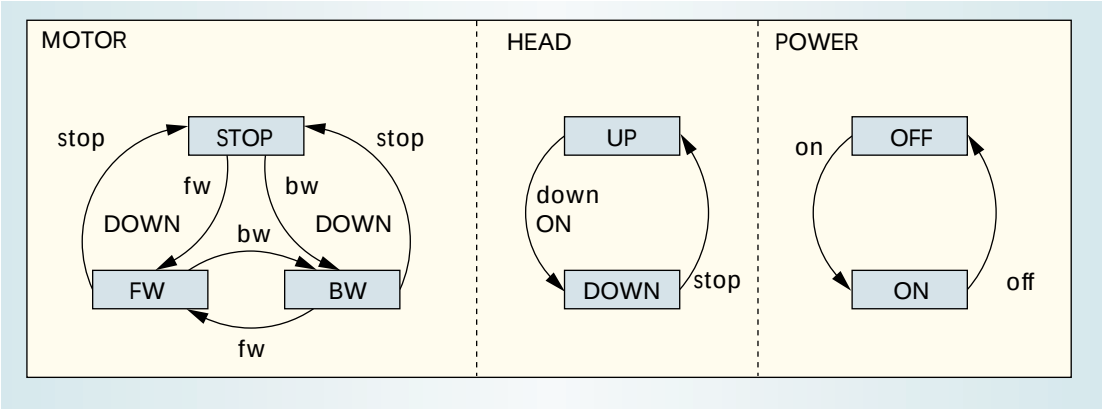

Figure 2. A simple model of a videocassette recorder. The model consists of three state machines: MOTOR, HEAD, and POWER.

or lifting the playing head, changing the rotational direction of the motor, and so forth.

CHECKING PROPERTIES

VisualState predefines seven generic properties that any well-formed state machine model should satisfy. The failure of a design to satisfy a particular check indicates a design error. For example, the tool determines whether the design has a deadlock, that is, a state in which none of the machines can react on any input event—clearly an undesired property in an embedded application. The tool also checks that any transition becomes enabled after some sequence of input events, and similarly, that any local state is entered after some sequence of input events. Transitions never enabled and local states never entered amount to dead code, which clearly ought not to be present in well-designed software.

The remaining four generic properties are

- no machine has a deadlock,
- no machine has two transitions from the same state that can be enabled simultaneously (corresponding to nondeterminism),
- all outputs are used, and
- all inputs are used.

All of these predefined checks address the problem of determining whether a given combination of local states is reachable. For example, the check that a given transition becomes enabled after some sequence of input events is reduced to a question of reachability by considering the local state from which the transition originates and the local states mentioned in the transition guard. If this combination of local states is reachable, a sequence of inputs enables the transition.

VisualState's design contributes to practical success because it checks from a fixed number of predefined properties rather than requiring the designer to choose and formalize properties. This characteristic makes the procedure automatic, like checking that a program is syntactically correct.

Our research takes a more general viewpoint regarding reachability checking, focusing on whether certain combinations of local states of individual machines can be reached after some sequence of input events.

VERIFICATION ALGORITHMS

We can use either forward or backward iteration to determine whether a given combination of states is reachable. Both of these techniques are fully automatic and complete in the sense that they always terminate and answer yes if and only if the state is reachable. However, their performance is radically different.

Implicit state representations

The reachable state space's size grows exponentially with the number of state machines in a design. Thus, even moderately complex embedded controllers have extremely large state spaces. This state-explosion problem quickly makes doing reachability checks by explicitly constructing the set of reachable states infeasible.

Instead of explicitly enumerating a design's reachable states, we can construct the state space implicitly. In this approach, Boolean predicates represent sets of states, and Boolean variables encode each local machine state. We can use the notation $M_i = s_j$ to represent a Boolean predicate stating that machine M_i is in the local state s_j. A Boolean function f (called the characteristic function for the set) then represents a set of states as follows: A state s is in the set of states represented by f if and only if $f(s)$ is true. Thus, a function f represents a set of states $\chi(f)$ given by

$$\chi(f) = \{s \mid f(s) = \mathbf{true}\}.$$

Notice that Boolean operations can perform the standard set operations on the characteristic functions for the sets. For example, set-union corresponds to a disjunction

$$\chi(f_1) \cup \chi(f_2) = \chi(f_1 \vee f_2).$$

In the videocassette recorder model shown in Figure 2, the Boolean function f given by

$$f = \neg(\textbf{MOTOR} = \text{STOP}) \wedge (\textbf{HEAD} = \text{UP})$$

represents the set of states

$$\chi(f) = \{(\text{FW},\text{UP},\text{OFF}),(\text{FW},\text{UP},\text{ON}),(\text{BW},\text{UP},\text{OFF}),(\text{BW},\text{UP},\text{ON})\}.$$

The central observation is that storing (and manipulating) Boolean predicates is often more efficient than representing the elements of the set $\chi(f)$.

Forward technique

Given a set of states S, the reachability problem is to determine whether the state $s \in S$ is reachable. The set S typically is given as a Boolean predicate defining some combination of the local states of the individual machines. For example, to determine whether the local state BW in machine MOTOR in Figure 2 is reachable, we check whether S given by the predicate (MOTOR = BW) is reachable.

A standard way to determine whether S is reachable is to first construct the set of reachable states and then examine whether S intersects this set. Assume that R is a Boolean function that represents the set of reachable states. To test whether a state S intersects R, you simply determine whether the Boolean function $R \wedge S$ is satisfiable.

In Figure 3, R_k describes the set of states that can be reached from s_0 with input sequences no longer than k. The function NEXT(R_k) determines the set of states that can be reached in one transition from a state in R_k. The set of reachable states R is obtained when the above iteration converges (that is, when $R_{k-1} = R_k$).

Compositional backward technique

You can also use a backward iteration to determine the reachability of a set of states S. Instead of starting with the initial state s_0, you start with the state S and use the iteration shown in Figure 4 to construct the set of states that can reach S (denoted B). The function PREV(B) returns all states that can reach a state in B in one transition. The state S is reachable if and only if the initial state s_0 is in B.

Notice that, compared with the forward iteration, backward iteration has an apparent drawback when performing many reachability checks: Instead of using a *single* iteration to construct the reachable state space R, each reachability question gives rise to a new backward iteration (because the set B cannot be reused). Our idea is to perform the backward iteration compositionally, involving a minimal number of machines.[6]

The compositional backward technique is based

FORWARD (s_0) =
$k := 0$
$R_0 := \{s_0\}$
do
$k := k + 1$
$R_k := R_{k-1} \cup \text{NEXT}(R_{k-1})$
while $R_{k-1} \neq R_k$
return R_k

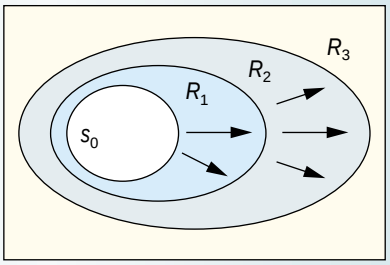

Figure 3. Using forward iteration to construct a set of states.

BACKWARD (S) =
$k := 0$
$B_0 := \{S\}$
do
$k := k + 1$
$B_k := B_{k-1} \cup \text{PREV}(B_{k-1})$
while $B_{k-1} \neq B_k$
return B_k

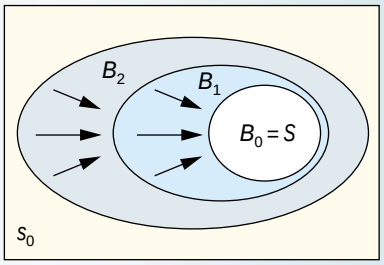

Figure 4. Using backward iteration to construct a set of states.

on the concept of one machine being dependent on another. A machine M is said to depend on M' if M has a transition with a guard referring to a local state of M'. A set of machines I is *dependency-closed* if for any machine M in I, all machines on which M depends are also in I. Consider the videocassette recorder model in Figure 2. Because machine HEAD has a transition that depends on the machine POWER, HEAD depends on POWER. The set of machines {HEAD, POWER} is dependency-closed because neither machine depends on the third machine, MOTOR.

The compositional backward technique iterates using only some subset, I, of the machines. Initially, I contains the subset of machines with a local state in S. Using the function PREV*(I,B), we determine all the states that can reach a state in B through transitions of the machines in I, making no assumptions about the local states of the machines not in I. This function is similar to the BACKWARD procedure in Figure 4 except that it starts from the set of states B, and the PREV function only considers machines in I.

If this set contains the initial state s_0, we can conclude that S is reachable. If the set does not contain s_0, we cannot in general conclude that S is unreachable because the set of considered machines I may be too small. Only when I is dependency-closed do we know that adding more machines to I will not have

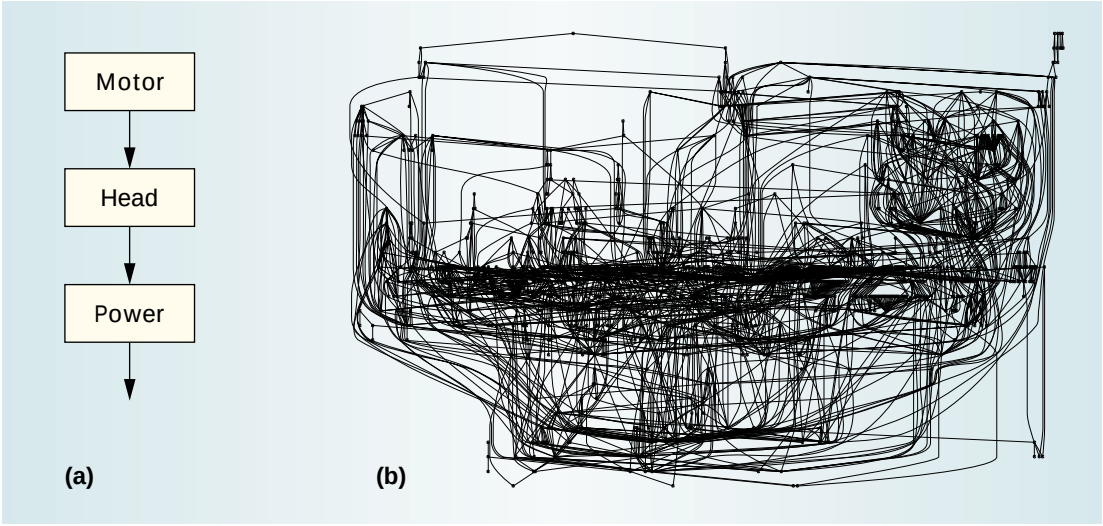

Figure 5. The dependency graph for (a) the videocassette recorder in Figure 2 and (b) for the Train2 state machine model. In the Train2 design, the largest dependency-closed set contains only 234 of the 1,421 machines.

an effect on the resulting set of states, and we can conclude that S is not reachable.

```
ISREACHABLE(S) =
  I := {M : machines mentioned in the
           predicate S }
  B := S
  repeat
    B_new := PREV*(I,B)
    if s0 ∈ B_new then return true
    if I is dependency-closed then
      return false
    Extend I with at least one
      machine on which a machine in I
      depends.
    B := B_new
  forever
```

The compositional backward technique has the following unique properties:[6]

- It is *compositional*. It involves as few machines as possible to settle a given reachability question.
- It is *incremental*. When it needs to include more machines, it reuses the already computed portion of the state space rather than having to start from scratch.
- It includes new machines as needed according to a *dependency analysis*.
- It allows *early termination* before inclusion of all machines. It concludes reachability as soon as the initial state is in the computed portion of the state space; it concludes nonreachability when the included machines constitute a dependency-closed collection and the initial state has not yet been encountered.

Using the dependency relation between the machines is essential for efficiently analyzing large-state machine models. Figure 5 shows the dependency graph for the videocassette recorder and the Train2 design, which has 1,421 state machines. Each vertex in the dependency graph represents a state machine, and an edge from one vertex to another indicates that one machine depends on another.

Table 1. The characteristics of six machine models.

Model	Number of state machines	Number of reachability checks performed	Declared size of state space	Reachable state space
Flow	10	240	10^5	17,040
AVS	12	176	10^7	1,438,416
Volvo	20	103	10^{11}	9.2×10^9
N8	111	837	10^{40}	—
Train1	373	1,664	10^{136}	—
Train2	1,421	6,046	10^{476}	

DESIGN TESTS

We applied the two techniques for performing reachability checks to a range of industrial designs covering various applications developed with visualState. To demonstrate the efficiency of these techniques, we selected the six machine models listed in Table 1. These examples range in complexity from small applications with as few as 10 state machines up to Train2, one of the largest industrial applications ever to be exhaustively verified.

We tested these designs to verify that they satisfy the seven generic properties that a well-formed state machine model requires. Each state machine model requires numerous checks, with the largest requiring

close to 20,000 checks. However, implicational analysis can substantially reduce the actual number of checks required.[11] For the applications we considered, implicational analysis eliminated between 40 percent and 94 percent of the checks.

Table 1 shows the reduced number of reachability checks. The size of the reachable part of the declared state space was computed from a forward iteration. Forward iterations cannot find the reachable state space for the two largest applications, and the size of these is unknown.

Table 2 displays runtime data for the experiments. The explicit forward column required 2 bits per state in the declared state space.

We used a standard PC (166-MHz Pentium PC with 32 Mbytes of RAM running Linux) to test these applications. For the implicit representation and manipulation of the state spaces, we used Reduced Ordered Binary Decision Diagrams (ROBDDs),[12] an efficient data structure for Boolean functions. The forward technique requires a single iteration to find the reachable state space, followed by a series of simple operations to determine the reachability checks. The compositional backward technique requires performing more costly iterations for each reachability check, but it uses only a subset of the machines to perform the iterations. Taking the large number of checks into account, it is therefore not clear how the two techniques will compare.

For comparison, we also performed the verifications using an explicit representation of the state space. The explicit representation requires 2 bits of memory for each state in the declared state space. The declared state space is the set of all possible states, computed as the product of the local states of each of the state machines. In contrast to ROBDDs, explicit representation is very sensitive to the size of the declared state space, thus it is not feasible for larger state machine models. Even a small state machine model with 10^9 states requires 256 Mbytes of memory. In contrast, the implicit technique performs all the verifications with modest use of memory (≤11 Mbytes).

As expected, the explicit technique succeeds only with the smallest applications. The implicit forward iteration is efficient for the four smallest designs, but fails on the two largest designs. When the ROBDD technique succeeds, it is fast, and it is superior to explicit state enumeration even for designs with a small number of reachable states.

For the largest state machine models, only the compositional backward technique succeeds. In fact, for the four largest designs, it is the most efficient technique. For small applications, its performance is comparable to the forward technique despite the high number of checks and the necessity to repeat backward iterations for each check.

Table 2. Runtime of machine model applications in CPU seconds.

Model	Number of state machines	Explicit forward technique	Implicit forward technique	Implicit backward technique	Implicit compositional backward technique
Flow	10	5s	0.3s	3.1s	2.2s
AVS	12	679s	2.2s	4.3s	2.4s
Volvo	20	—	1.8s	2.0s	0.7s
N8	111	—	—	666.0s	39.3s
Train1	373	—	—	480.6s	11.1s
Train2	1,421	—	—	—	271.7s

— = Analysis depleted memory or runtime exceeded two hours without finishing.

To understand why the compositional backward technique is successful, we analyzed the Train2 design in more detail. The largest dependency-closed set contains 234 machines. Thus, no more than 234 machines out of a possible 1,421 are ever included in the verification. In fact, except for an unsuccessful check, fewer than 32 percent of the machines in a dependency-closed set are ever needed.

GOING FURTHER

A hierarchical extension of the state machine model describes complex applications more concisely. Since its introduction in 1987 as the pioneer in hierarchical descriptions, designers have accepted Statecharts[2] as a compact, practical notation for reactive systems. A number of hierarchical specification formalisms are now available, including the emerging Unified Modeling Language.[1]

In a hierarchical state machine, the set of states constitutes a nesting structure, and each state is either a primitive or superstate containing a set of state machines. To obtain an equivalent nonhierarchical design in a hierarchical state machine, you can flatten it. Flattening recursively introduces each superstate's associated state machine as a concurrent component. Although flattening preprocessing is a conceptually simple approach to verification, a state machine model's hierarchical depth limits the effectiveness of this approach. Flattening increases the number of concurrent components, and all of the enclosing superstates need to guard the transitions within the newly generated machines. The high degree of dependencies between these components degrades the compositional backward technique's performance.

To address this problem, we extended the compositional backward technique to exploit its hierarchical structure by reusing earlier superstate reachability checks to determine substate reachability.[7] Figure 6 shows how the flattening approach degrades perfor-

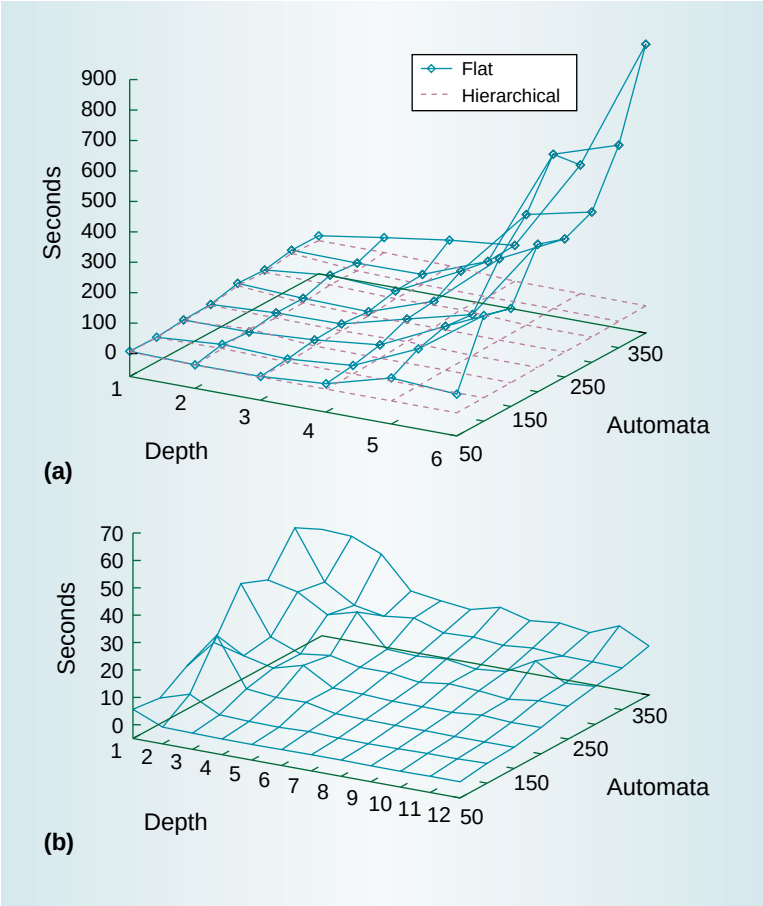

Figure 6. Comparison of flat and hierarchical checker runtimes: (a) runtime of both checkers at nesting depths ranging from 1 to 6; (b) the hierarchical checker runtime at nesting depths ranging from 1 to 12. Each grid point represents the mean runtime of 20 test cases, each generated using the same parameters.

mance with increased hierarchical depth; in contrast, the extended approach is insensitive to hierarchical depth, and the performance even improves as the depth increases.

Implementing the compositional backward technique with newer visualState versions has drastically increased the size of the state machine models that it can verify in industrial applications. However, because we can expect the execution time and memory requirement of any exhaustive verification algorithm to grow exponentially with the design's size, we need to evaluate a verification technique on the basis of its ability to solve problems in real applications.

Although the compositional backward technique is very successful in this respect, we did encounter a problem with a compact zoom-camera state-machine model. Even though this model contains only 36 state machines, we could not find a technique that could fully verify this application. This emphasizes the point that simply counting the number of machines does not provide a true measure of a verification task's complexity. Additional research is needed to more accurately predict the difficulty of verifying a given design with a given technique. ✸

Acknowledgments

This work was supported by the Danish National Center for IT Research.

References

1. G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide*, Addison-Wesley, Reading, Mass., 1997.
2. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, June 1987, pp. 231-274.
3. N.G. Leveson et al., "Requirements Specification for Process Control Systems," *IEEE Trans. Software Engineering*, Sept. 1994, pp. 694-707.
4. J.R. Burch, E.M. Clarke, and K.L. McMillan, "Symbolic Model Checking:10^{20} States and Beyond," *Information Computation*, June 1992, pp. 142-170.
5. J. Lind-Nielsen and H.R. Andersen, "Stepwise CTL Model Checking of State/Event Systems," *Computer-Aided Verification*, Springer-Verlag, Berlin, 1999, pp. 316-327.
6. G. Behrmann et al., "Verification of Hierarchical State/Event Systems: Reusability and Compositionality," *Proc. Fifth Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems*, Springer-Verlag, Berlin, 1999.
7. J.R. Burch et al., "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Apr. 1994, pp. 401-424.
8. W. Chan et al., "Model Checking Large Software Specifications," *IEEE Trans. Software Eng.*, July 1998, pp. 498-520.
9. K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic, Boston, 1993.
10. G.H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Technical J.*, May 1955, pp. 1045-1079.
11. J. Lind-Nielsen et al., "Verification of Large State/Event Systems Using Compositionality and Dependency Analysis," *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Mar./Apr. 1998, pp. 201-216.
12. R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, Vol. 8, No. C-35, 1986, pp. 677-691.

Jørgen Staunstrup is head of the Department of Hydroinformatics at DHI, Hørsholm. He earned a PhD in computer science from the University of Southern California and a Dr. Techn. from the Technical University of Denmark. Contact him at jst@dhi.dk.

Henrik Reif Andersen is an associate professor of computer science at the IT University of Copenhagen. His research interests include automatic formal verification of concurrent and reactive systems, and modeling, programming, and testing of embedded software. He received a PhD in computer science from Aarhus University. Contact him at hra@itu.dk.

Henrik Hulgaard is an associate professor of computer science at the IT University of Copenhagen. He is interested in timing analysis and verification of asynchronous circuits and real-time systems and in formal

verification of concurrent systems, with particular emphasis on the analysis of embedded software. He received a PhD in computer science from the University of Washington. Contact him at henrick@itu.dk.

Jørn Lind-Nielsen is a PhD candidate in computer science at the IT University of Copenhagen. His research interests include techniques for automatic verification. He received an MSc in electrical engineering from the Technical University of Denmark. Contact him at jl@itu.dk.

Kim G. Larsen is a professor of computer science at Aalborg University. His research interests are concurrency theory, specification, design, construction, validation, verification and testing of embedded systems, and real-time and hybrid systems. He received a PhD in computer science from Edinburgh University. Contact him at kgl@cs.auc.dk.

Gerd Behrmann is a PhD candidate in computer science at Aalborg University. His research interests include techniques for automatic verification of hierarchical and real-time systems. He received an MSc in computer science from Aalborg University. Contact him at behrmann@cs.auc.dk.

Kåre Kristoffersen is a research assistant professor in the Distributed Systems and Semantics Group in the Department of Computer Science, Aalborg University. His research interests include the development of automatic tools for carrying out verification of concurrent systems. He received a PhD in computer science from Aalborg University. Contact him at jelling@cs.auc.dk.

Arne Skou is an associate professor of computer science at Aalborg University. His research interests include models for concurrency, tools for verification and validation, and application of formal methods on real-time industrial systems. He received a PhD from Aalborg University. Contact him at ask@cs.auc.dk.

Henrik Leerberg is executive vice president of IAR Systems A/S. He is responsible for managing all software research and development activities and customer services. Contact him at hleerber@baan.com.

Niels Bo Theilgaard is general manager for Baan's front office activities. He received a PhD in systems science from the Technical University of Denmark. Contact him at nbtheilgaard@baan.com.


XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX