Program Verification

Robert P. Kurshan

How can a computer program developer ensure that a program actually implements its intended purpose? This article describes a method for checking the correctness of certain types of computer programs. The method is used commercially in the development of programs implemented as integrated circuits and is applicable to the development of "control-intensive" software programs as well. "Divide-and-conquer" techniques central to this method apply to a broad range of program verification methodologies.

Classical methods for testing and quality control no longer are sufficient to protect us from communication network collapses, fatalities from medical machinery malfunction, rocket guidance failure, or a half-billion dollar commercial loss due to incorrect arithmetic in a popular integrated circuit. These sensational examples are only the headline cases. Behind them are multitudes of mundane programs whose failures merely infuriate their users and cause increased costs to their producers.

A source of such problems is the growth in program complexity. The more a program controls, the more types of interactions it supports. For example, the telephone "call-forwarding" service (forwarding incoming calls to a customer-designated number) interacts with the "billing" program that must determine whether the forwarding number or the calling number gets charged for the additional connection to the customer-designated number. At the same time, call-forwarding interacts with the "connection" program that deals with the issue of what to do in case the called number is busy, but the ultimate forward destination is free. One property to check is that a call is billed to the customer if and only if the connection is completed. If the call connection and billing programs interact incorrectly, a called number that was busy and then became free could appear busy to one program and free to the other, resulting in an unbilled service or an unwarranted charge, depending upon their order of execution.

If a program includes n interrelated control functions with more than one state, the resulting program may need to support 2^n distinct combinations of interactions, any of which may harbor a potential unexpected peculiarity. When n is very small, the developer can visualize all the combinations and deal with them individually. Since the size of a program tends to be proportional to the number of functions it includes (one program block per function), the number of program interactions as a function of program size may grow exponentially. As a result, the developer can use only a very small proportion of the possible program interactions to guide the development and testing of the program. When some unconsidered combination produces an eccentric behavior, the result may be a "bug".

While a computer could be programmed to check a program under development for eccentric behavior by searching exhaustively through all combinations of program interactions, the exponential growth could soon cause the computer to exceed available time or memory. On account of the potential for interactions, adding a few functions to a program can substantially alter its testability and thus viability. From the program developer's perspective, this is unsatisfactory. If the program is developed carefully, however, the correct behavior of each of the n individual control functions of the program can be checked in a way

Robert P. Kurshan is Distinguished Member of Technical Staff at Bell Laboratories, Murray Hill, NJ. His e-mail address is `k@research.bell-labs.com`.

An expanded version of this article is appearing in the Birkhäuser series Applied and Computational Control, Signals, and Circuits planned for publication during 2000.

such that the complexity of the checks grows more or less proportionally to the size of the program.

Overview

This article presents some key ideas of "program verification". It focuses on the "reduction and decomposition" process, which addresses the problem of how to verify a program automatically in a way that the computational complexity scales tractably with increasing program size. It does not attempt to survey the field. A survey of automaton-based verification is included in [9]; for a survey of logic-based verification, see [4], [7].

Reduction and Decomposition

Reduction and decomposition circumvent the exponential cost of checking correctness. Checking each control function separately, *reduction* refers to an algorithm[1] through which the program to be checked is replaced, relative to the respective control function, with a simpler program that it is sufficient to check. For example, to check the operation of call-forwarding, the part of the program that performs billing may be largely ignored, except to the extent that it may interfere with call-forwarding. The reduced program is checked through an algorithm that analyzes every possible program state. In order for a computer to implement this effectively, the reduced program must be sufficiently simple.

Decomposition refers to checking that a given control function is correctly implemented by splitting the function into several simpler-to-check "subfunctions". Taken together, the subfunctions implement the original function. A subfunction is simpler to check if it gives rise to a simpler reduction than the original function. Decomposition and reduction thus are used together as divide-and-conquer techniques. As an example, call-forwarding may be decomposed into its "setup", where the customer designates the new destination for incoming calls, and "execution", where an incoming call gets forwarded to the designated destination. In verifying setup, the reduced program can, in addition, ignore most of the parts of the original program that perform the execution subfunction. Likewise in the verification of the execution subfunction, most parts of the program that deal with setup can be ignored. A computer program can check that these subfunctions collectively implement the original function. In general, it is a manual step to decide how to decompose a function.

Conventional Testing vs. Program Verification

Suppose one wanted to check the C program in Figure 1, whose stated purpose is to read a non-negative binary integer from standard input and convert it to a decimal integer printed to standard output. For example, entering "11001" followed by a carriage return should produce "25".

```
#include <stdio.h>
main()
{
  unsigned x=0;
  int input;
  while((input=getchar())!=EOF) {
    if(input=='0')x=x*2;
    else if (input=='1')x=x*2+1;
    else if (input=='\n') {
      printf("%u\n",x);
      x=0;
    }
  }
}
```

Figure 1. A program for converting binary to decimal.

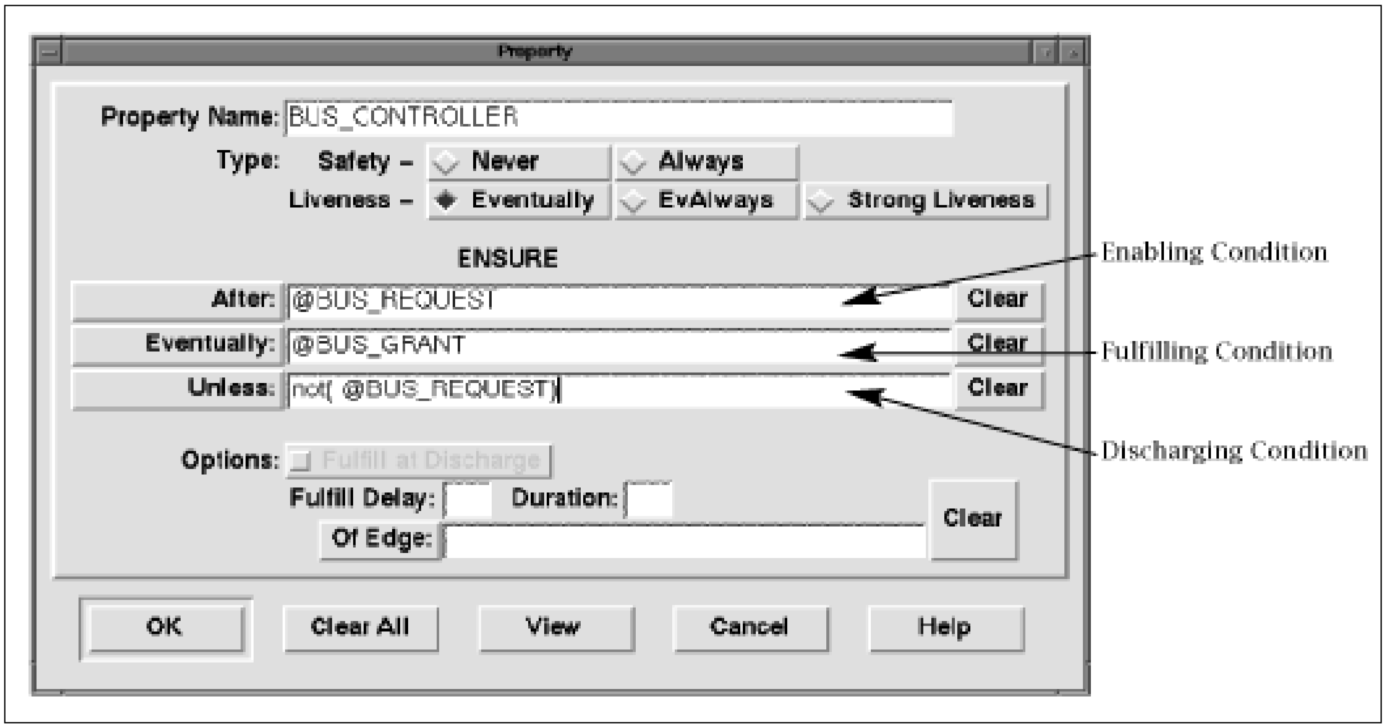
Program verification could demonstrate that this program fails to fulfill its stated purpose. Readers who are C programmers are invited to stop reading this article now and find the bug in this trivial program.

In fact, the bug is nowhere to be found in the text of the program. Computers commonly truncate "integer" variables to 32 bits (in binary representation), thus performing integer arithmetic modulo 2^{32}. In the application of program verification, this truncation would be inferred from knowledge of the type of computer on which the program is run. The catastrophic failure of the European Space Agency's Ariane 5 Flight 501 has been traced to a numerical overflow like the one in this example.

To check the program completely for the stated purpose, one would need to confirm that it is fulfilled at every program state. If the program is tested for billions of "typical" inputs, all of which happen to have fewer than 32 bits, the bug would be missed.

The program above is utterly trivial, giving no inkling of the complexities to be found in routine programs, which could be hundreds of pages long. Nonetheless, if one tried to test this program for the stated purpose by executing the program from the keyboard, entering the binary integers consecutively, it would take about one hundred years to hit the bug. This time could be reduced to a couple of hours by writing an automated tester. For a program not so trivial, even an automated tester would be unable in any lifetime to execute the program under test to reach all its states. The automated tester would need to make "educated guesses" at input sequences likely to uncover errors. Since programming errors tend to be unintentional, they can hide in hard-to-guess corners of a program.

[1]*An algorithm is a precisely specified succession of steps to obtain a desired result, such as Euclid's gcd algorithm. In this article the term refers to a procedure that generally is an* automatic *step in the verification process (a step implemented by a computer program).*



Property Name: BUS_CONTROLLER

Type: Safety – Never / Always

Liveness – [✓ Eventually] / EvAlways / Strong Liveness

ENSURE

After: @BUS_REQUEST

Eventually: @BUS_GRANT

Unless: not(@BUS_REQUEST)

Options: Fulfill at Discharge

Fulfill Delay: Duration:

Of Edge:

Figure 2. In the commercial model-checking tool FormalCheck [11], required program operation is specified in terms of an *enabling condition* that determines the onset of a requirement specified by a *fulfilling condition* that must hold unless released by a *discharging condition*, ad infinitum. Each such specification is represented internally by an ω-automaton. Any property expressible with an ω-automaton can be expressed by a collection (logical conjunction) of such specifications.

Tailoring a tester to the program under test is a nontrivial programming burden that adds significantly to the cost of program development and delays the onset of testing. This task can be simplified by generating inputs randomly. While easier (and faster) to set up, random testing often is nearly worthless. For the example above, if input strings of 0's, 1's, and carriage returns were randomly generated with respective probabilities .25, .25, and .5 (biased in favor of "typical" shorter input strings), the expected time to hit the bug would be more than the time required to execute the program for each successive input string with value increasing from 0 to 2^{32}. For less trivial programs the probability of hitting a bug in a reasonable amount of time with randomly generated inputs may be close to 0. For each less trivial program to be tested, the "problematic" regions of its state space must somehow be predicted, and a "test bench" must be built to guide the tests to these regions and capture errors.

By contrast, a program verification algorithm can find this bug in a few seconds on a modern computer. No problematic regions need be predicted, and no test bench need be built. Instead, it is necessary only to give a precise definition of what is to be checked. For the example program, this can also be done in a few seconds using a user-interface aid like the one pictured in Figure 2 to specify the property that the decimal output always is equal to the binary input. To express this property, a user of the Figure 2 aid first sets the Type to *Always*, causing the qualifier of the Fulfilling Condition to change from *Eventually* (as shown) to *Always*, and then the user enters the Fulfilling Condition that expresses in program variables the condition `decimal (inputbits) = output` (leaving the other two fields blank). The user interface aid will automatically generate an "automaton" that defines this property. This automaton is described below. (Hardware verification tools such as [11] operate on hardware description languages instead of C code, but the principle is the same.)

If the purpose of the example program is modified, requiring that it work only for input strings of length 32 or less, then the program can be verified to be correct through an algorithmic check, described below, that takes a few seconds as well.

Conventional testing with either selectively generated input scenarios or random inputs can determine what happens only in selected cases. Especially in case of the presence of unknown bugs, a program's state space may not conform to expectations. A definitive test would need to include every possible input sequence. Since general input sequences are unbounded, there are an infinite number of sequences that should be tested. Executing the program for an infinite number of input sequences is not possible, so testing such a program by executing it is intrinsically incomplete.

Formal Verification

Program verification is also called "formal verification" to contrast it with conventional testing, which lacks a precise description of what was tested. Formal verification begins with a formal statement of some high-level purpose of the program and determines whether that purpose is supported in every possible execution of the program. This is accomplished by analyzing the logic of the program, not by executing it. Since formal verification can account for every possible program execution, it is more reliable than conventional testing. There are numerous accounts of bugs that could have caused significant commercial damage had they remained but that were found quickly with formal verification after having been missed by conventional testing.

As a program's purpose evolves, the influence of one part on another grows, so correcting defects becomes increasingly costly. A common observation is that the cost of fixing a bug doubles with each successive stage of development. Finding and correcting program errors early in the design cycle thus can decrease development time and cost significantly. Conventional testing requires the creation of program-specific test benches, so it tends to be deferred until late in the program development cycle, when the program development has stabilized. By contrast, formal verification algorithms are independent of the program to be verified. Therefore, formal verification can be applied early in the program development cycle.

Within the last ten years research in formal verification has found its way to commercial products that implement program verification algorithms. For the present these products are specialized mainly for industries that manufacture integrated circuits. However, the program verification methodologies described here can be applied to significant portions of general software development, and this is the direction of the industry.

Verification Methodologies

Formal verification refers not just to a single methodology but to a spectrum of methodologies. In this spectrum there is a tradeoff between *expressiveness*—what program properties can be described and analyzed—and *degree of automation*—the extent to which the verification can be automated with an efficient algorithm.

The most expressive form of formal verification could be said to be mathematical argument. However, the reliability of mathematical argument is based upon peer review [5]. It is unlikely that too many peers could be mustered to check a 100-page proof of the correctness of an integrated circuit. Moreover, it is unlikely that the circuit manufacturer would be willing to wait for the years it might take to produce such a proof in the first place.

This latter problem is shared by automated theorem-proving [2]. The dream of writing a computer program capable of automatically proving marvelous theorems never was completely realized. Instead, the "automatic theorem provers" are in fact proof checkers. While often quite sophisticated, powerful, and useful, automatic theorem provers mainly require the user to come up with the proof, which must be entered into the "theorem-proving" program using the special notation accepted by the program. Theorem provers thus do not at present support a methodology that can keep up broadly with a commercial development cycle.

This disappointment led researchers to investigate the other end of the spectrum: methodologies that sacrifice expressiveness in favor of automation. Among the first to discuss this strategy were E. M. Clarke and E. A. Emerson, who, in 1980, proposed a limited but completely algorithmic form of verification they called "model checking" [3]. Their mode of verification was founded in a logic that supported a very simple model-satisfaction check. Around the same time and independently, J.-P. Quielle and J. Sifakis made a similar proposal.

In 1982, ignorant of the above work, I proposed a verification method based on automata. Eventually (ten years later!) the method was implemented in a commercial tool (Figure 2) that is marketed to manufacturers of integrated circuits. Since for the moment at least this is the dominant commercial model-checking tool, I will exercise a certain prerogative and restrict my discussion of model-checking to automata. There is no loss of generality in using automata, as logic-based model checking can be reduced to automaton-based model checking (and vice versa).

Models

There is a substantial literature reporting the impressive benefits of formally verifying systems for everything from communication to safety-critical devices. It may come as a surprise then (cf. [1]) that there is no mathematical sense in which these can be verified. The thinking is that the mathematical model of a computer system can be the exact and entire computer program itself. That is, the computer's programming language may be given a formal semantics,[2] and one may then reason formally about the actual program. So it may seem that if the formal model is the entire system program, then any formal proof about the behavior of the model must be a proof about *the actual system*, since they are one and the same.

[2]*A computer programming language has the formal attributes of "syntax" and "semantics".* Syntax *is a set of rules, such as those checked by a program compiler, governing when a string of characters defines a "well-formed" expression.* Semantics *is a map of syntax to a well-understood mathematical model, such as an automaton, which gives meaning to a computer program. The semantics masks both "uninteresting" variations in syntax and "irrelevant" details of the computer program execution, such as transient states ignored by the computer operating system. Program semantics gives a basis to determine whether two computer programs are "essentially" equivalent.*

The fallacy is that a computer program by itself embodies no physical behavior. Physical behavior is manifest when the program is entered into a computer and executed. The observable physical behavior thus is not of the computer program but of the computer. The computer's behavior is related to its hardware, to whether the computer is plugged in and operated properly, and to the nature of other programs, such as the compiler. Conceptually (but definitely not realistically), mathematical models of all these could be taken into account too. Physical models of even wires and transistors are imperfect predictors, however, depending on uncertainties like model inaccuracies, manufacture process, and defects. The important point here is that there is no *absolute* value in formally verifying a program, such as a guarantee that its execution on a physical computer will behave correctly.

Moreover, an insurmountable deficiency of formal verification and conventional testing alike is the inability to know that the operational definition of "behave correctly" is "correct". My computer has two implementations of the Unix "word count" utility $wc(1)$. They disagree on whether a final string terminated without white space or a newline counts as a word. In the fall of 1999 another incompatibility in software minutiae led to the destruction of NASA's Mars Climate Orbiter. Even had the derelict NASA software been formally verified module by module, the mismatch in measurement units between modules could have gone unnoticed unless the entire system had been verified as a single entity. To do this may have been impossible within the time available. If the navigation table, erroneously expressed in English units rather than metric units, was preexisting or otherwise deemed "external" to the verification process—and one must draw the boundary somewhere—then the mismatch in any case could have been overlooked.

While all this uncertainty may seem to cast a cloud over the entire enterprise of program verification, the uncertainty should be taken in perspective. There can be no absolute guarantee that a program will behave correctly, even in principle. As long as a verification process improves the reliability of programs or shortens the program design cycle more than other methods, it is worthwhile. Model checking turns out, above all, to be an uncommonly effective debugging tool.

To apply a program verification algorithm to the C program in Figure 1, the first step is to translate the C program to a simple representation with a clear semantics, like automata described below, on which the verification algorithm has been designed to operate. Commercial model checkers use a general automated *translator* to do this for "suitable" programs. Not all C programs are amenable to algorithmic verification. The C language supports unbounded constructs like character pointers that can be incremented to point to an unspecified range of memory locations. Each location in effect defines (an unnamed) program variable, and the possible extent of this is determined by the memory configuration of the particular computer on which the program is run.

The synergy between programs that define integrated circuits and program verification derives in part from the fact that programming restrictions sufficient to derive an automaton model from a program—a finite number of states, and a well-defined state-transition structure—are necessary to generate an integrated circuit automatically from a program. Unlike most C programmers, circuit designers thus intrinsically write programs that also can be verified. Nonetheless, many portions of C programs like the example in Figure 1 are finite-state and thus can be translated automatically to an automaton model. An infinite-state C program that implements a "control" function like call-forwarding typically has core components that can be formally abstracted to a finite-state program. It is an open problem of software engineering to identify such translatable portions automatically.

For translatable programs the translation process is fraught with pitfalls. A given input program may compute different results on different computers. For example, incrementing an integer variable on a computer with a 32-bit or a 64-bit architecture is implicitly modulo 2^{32} or 2^{64} respectively. There are myriad unpublished conventions concerning the transparency to the user of transient program states, and yet there is no universal convention concerning what constitutes a stable program state. All such issues must be covered by the translator.

Program State

In the von Neumann stored program model, "instructions" update "data" stored in "memory registers" in a sequential fashion. "Program variables" designate memory locations and their data contents and also define via "assignment expressions" the rules by which the data stored in the variables get modified.

The *program state* is the vector of simultaneous values of all program variables. It captures the entire instantaneous condition of the program, independent of its history. A program's *behavior* is captured by its succession of states. However, the computer or integrated circuit that implements the program makes it effectively impossible (on purpose)—or renders it unnecessary—to discern the precise sequential evolution of certain "transient" states. The program model should reflect this and moreover can exploit it to reduce the computational complexity of verification. For example, for variables `x`, `y`, and `z`, suppose `x` is assigned to take the value of `y+1` after `y` is assigned to take the value of `z+1`. An analysis of the program may reveal that with regard to its implementation, `x` effectively is or can be treated as a "macro" or synonym for `y+1` and likewise y for z+1, simplifying the two successive assignments to a simultaneous assignment of the value of z+1 to y and of z+2 to x.

Deciding which assignments are to be modeled as simultaneous must be done with great care in order to preserve the semantics of the implemented program. Commercial model checkers perform this task algorithmically. In the program model a *sequential* variable is one that attains its value after (rather than simultaneously with) the value attained by its assignment expression. Typically, if a variable x is assigned the value of an expression that depends on x—for example, if x is assigned the value of x+1—then x would be designated a sequential variable. The new value of x would then be modeled as succeeding the prior value of x.

Limiting the number of sequential variables in a program is very important for the performance of an integrated circuit and for the performance of model checking too. There is a multibillion-dollar industry for "synthesis" tools that help create an integrated circuit layout automatically from its defining program, focused in important part on this partitioning problem.

The nonsequential program variables are called *combinational* variables (to use the hardware term). These hold the values of respective "transient" steps in a computation, such as the value of y used in x above. They include the *primary inputs*, which are variables whose values are assigned according to decisions made outside of the program, e.g., which keyboard key a human pushes. Typically, program outputs also are represented by combinational variables.

In the program model the program state is partitioned into the *sequential state,* defined by the values of the sequential variables, and the *combinational state,* defined by the remaining (combinational) variables. The combinational state vector $\mathbf{C}$ is given as a function $\mathbf{C} = \mathbf{f}(\mathbf{S})$ of the sequential state vector $\mathbf{S}$. The sequential state $\mathbf{S}$ is initialized, and each successive value

$$\mathbf{S}' = \mathbf{F}(\mathbf{S}, \mathbf{C}) \tag{1}$$

is expressed in terms of its current value $\mathbf{S}$ and the combinational state.

Nondeterminism

In a model of the program the assignment of primary inputs must be abstracted to account for all possible assignments. Abstraction is used more generally in program verification to simplify parts of a program whose detailed function is irrelevant to the verification objective. Abstracting duration in respective asynchronous parallel processes can result in a simpler model that is useful when the required behavior is independent of the relative execution speeds of the processes.

An effective way to abstract a program is through the use of *nondeterminism* in program variable assignments. If the function $\mathbf{f}(\mathbf{S})$ is allowed to be multivalued, at each sequential state $\mathbf{S}$ a combinational variable may assume several values in its range. This gives rise to a set of program behaviors or "lifetimes". Each nondeterministic assignment splits the execution of the program model into separate respective behaviors, ad infinitum.

Without nondeterminism a program would have only one behavior, and its analysis would be simple. Although it is the nondeterminism that makes program analysis difficult, there is no real alternative with regard to the primary inputs. It also may be simpler and sufficient to model with a nondeterministic assignment certain variables that *are* assigned within the actual program. Imagine a program subroutine that performs a complex computation producing a nonzero result, writing the result to the program variable v. Suppose the program is required to perform some task unrelated to the computation each time the subroutine completes its calculation, the completion indicated by $v \neq 0$. In order to verify the property P that whenever $v \neq 0$ the task is correctly performed, the computation that assigns v is irrelevant. To verify P, it is sufficient to have the program model assign v nondeterministically to 0 or 1, where v=1 is an abstraction that stands for all the nonzero values of the variable v in the original program. In this abstraction, the verification routine can determine, through a localization reduction described below, that the complex computation is irrelevant and exclude it from the analysis leading to the verification of P. (That computation will be relevant to other properties, with respect to which other portions of the program may be irrelevant.) If P is verified in the abstract model and the computation that assigns v in the original program is subsequently altered, it is unnecessary to reverify P.

If $w \neq 0$ signals the conclusion of another program, then the order in which v and w assume nonzero values indicates the relative speeds of the associated programs. Assigning v nondeterministically allows it to become nonzero both before and after w does. A property verified with this abstraction will hold for alternative implementations that vary the relative speeds of the programs.

Nondeterministic assignment may introduce additional behaviors not present in the original program (while retaining all original behaviors). In case several variables are assigned nondeterministically, not all combinations of assignments may be possible in the original program. On the other hand, nondeterministic assignment can reduce program complexity by masking relative execution speeds and program structure (as above). This makes it a vital tool in program verification.

Automata

The automaton is a fundamental structure in computer science, useful for much analysis and proof concerning "regular" sequential behavior. In the translation of a program to an automaton, defined next, the program's sequential state becomes the automaton "state", and the program's combinational states define automaton state transition conditions. For the C program in Figure 1, the states of the corresponding automaton are the 2^{32} values of the program's sequential variable `x`. The program's combinational variable `input` defines the state transition conditions as follows. For each value of `x` there are transitions to the states

$$(2)\qquad 2\mathtt{x},\ 2\mathtt{x}+1,\ 0$$

and back to `x`, conditioned respectively on `input` = 0, `input` = 1, `input` = `\n` (carriage return), and every other value of `input`. Program behaviors are modeled by the sequences of combinational states consistent with successive automaton state transitions (cf. a discrete Markov process). The set of all such sequences forms the *language* of the automaton, which provides the basis for model checking as defined here. The combinational states of the C program in Figure 1 are its respective input/output pairs (`input`, *output*), where *output* is the value imparted by the print statement (namely, the value of `x` , or "nothing", encoded by some distinct designated value).

To facilitate reduction and decomposition, the modeling automaton is "factored" into component automata, in emulation of the program's modularity. This factoring requires a means to correlate factor automata transition conditions that mediate the respective automaton state transitions. For example, if one program component reads the value of a variable set by another program component, the corresponding automata must have a mechanism to reflect a common view of the value of that variable. For reduction there also must be a means to "abstract" transition conditions. If a state transition is conditioned on a variable `v` being nonzero and the nonzero values of `v` are abstracted to the value "1" as above, then the corresponding automaton transition condition must be abstracted in a consistent well-defined manner. To meet these needs, the set of valuations of combinational program variables is given the structure of a Boolean algebra.[3] In this context the correlation of transition conditions is captured by their conjunction in the Boolean algebra, and abstraction is obtained through a Boolean algebra homomorphism. The principal role of the automaton defined next is to serve as a "scaffolding" to carry this Boolean algebra: the factorization (5) needed for the decomposition (6) appears as a matrix tensor product on the automaton state transition matrices, and the simplification needed for reduction is given as a Boolean algebra homomorphism that acts on these matrices elementwise, in (12) below. The details are now explained.

In the context of program verification the most useful type of automaton is the *ω-automaton*, whose behaviors are (infinite) sequences. The automaton is defined in terms of a directed graph with a finite set of vertices called *states*, some of which are designated *initial*; a *transition condition* for each directed edge or *state transition* in the graph; and an *acceptance condition* defined in terms of sets of states and state transitions. Each *transition condition* is a nonempty set of elements called *letters* drawn from a fixed set called the *alphabet* of the automaton. There are several different equivalent definitions of acceptance conditions in use. The acceptance condition of an ω-automaton can capture the concept of something happening "eventually" or "repeatedly" (ad infinitum), "forever" or "finally" (forever after). The set of all subsets of the alphabet forms a Boolean algebra L in which the singleton sets are atoms. The set of atoms of L, denoted by $S(L)$, determines L when L is finite.

How are automaton transition conditions associated with the program's combinational states? If v and w are states of the automaton A, let $A(v,w)$ denote the transition condition on the directed edge (v,w). Expressing conjunction (set intersection) in L by $*$, we see for a letter a that $a \in A(v,w)$ if and only if the atom $\{a\}$ satisfies $\{a\} * A(v,w) \neq 0$. In the context of L, we refer to the atom $\{a\}$ as a "letter" and $S(L)$ as the "alphabet" of A. We say the automaton A is *over* L. L is associated with the Boolean algebra generated by all valuations of combinational program variables: terms of the form $\mathtt{x} = c$ for a value c in the range of the combinational variable `x`. Thus, L is all Boolean expressions in these terms. Each letter is a conjunction of terms of the form $x = c$, the conjunction over all the combinational variables x. This corresponds to the combinational state $\mathbf{C}$ with x-th component c. Thus, we associate combinational states with letters, i.e., atoms of L. Automaton transition conditions are nonzero elements of L. For example, a transition condition $\mathtt{x} = 0$ corresponds to the set of all letters (i.e., disjunction of atoms) of the form $\cdots * (\mathtt{x} = 0) * \cdots$. Referring to (1), we have

$$(3)\qquad A(v,w) = \sum_{\mathrm{F}(v,\mathbf{C})=w} \mathbf{C},$$

the summation expressing disjunction in L.

[3]*A* Boolean algebra *[8] is a set with commutative, associative, idempotent binary operations* conjunction *and* disjunction *and the unary operation* negation, *which satisfy DeMorgan's Law; the set contains a universal element and its negation (called* zero*) and satisfies the natural relations for these. Every Boolean algebra can be represented by a set of subsets of some set—with the operations intersection, union, and set complement—and universal element consisting of the set. An* atom *is a nonzero element that is minimal in the sense that its conjunction with any nonzero element is itself or zero.*

A sequence (v_i) of states of A is a *run* of a sequence of letters $(s_i) \in S(L)^{\mathbb{N}}$ provided v_1 is an initial state and for all i, $s_i * A(v_i, v_{i+1}) \neq 0$. A run is *accepting* if its infinitely repeating states and transitions satisfy the automaton acceptance condition.

The set of sequences of letters with respective accepting runs in an automaton A is the *language* of A, denoted $\mathcal{L}(A)$.

The stated purpose of the C program in Figure 1 can be expressed by a 2-state automaton P that uses its state to remember whether the current output is the decimal equivalent of the last binary input. The state of P can be represented with the program variable y, having initial state $\mathsf{y} = 0$ and transition condition for the state transition from $\mathsf{y} = 0$ to $\mathsf{y} = 1$ represented by the expression in program variables that expresses the failure

(4) $\quad \mathsf{decimal(inputbits)} \neq \mathsf{output}$

The acceptance condition accepts runs that remain forever in the state $\mathsf{y} = 0$.

Program Factorization

The automaton model of a program is "built" from smaller automaton models of program components. A program component may comprise a single variable or a group of closely related variables. The program consists of its components. For example, the call-forwarding program may be implemented by respective components that define its setup and execution, as described earlier. A program is translated component by component to respective "component" automaton models. The component automata are combined as a "product" to form a single automaton that models the original program. This "factoring" of the program model into components follows the natural program structure and aids in translation as well as in reduction and decomposition. All component automata are over one fixed Boolean algebra L determined by the program.

The combinational variables of a program component may be interpreted as the "outputs" of that component. The simultaneous valuations of the outputs of a program component get translated to elements of L that generate a subalgebra of L, the "output subalgebra" of the corresponding component automaton. (In this context, program primary inputs translate to outputs of trivial automata.) The call-forwarding setup component p may have as outputs the variables m and n that designate the called and forward numbers respectively. If there are no other outputs of p, then the output subalgebra of the automaton that models p has as its atoms all expressions of the form $(\mathsf{m} = m_0) * (\mathsf{n} = n_0)$, for m_0, n_0 in the range of m and n respectively. The (interior) product of the respective output subalgebras of all the component automata is L. Each letter (atom of L) is a conjunction of atoms from the respective output subalgebras.

A program component is modeled by an *L-process* P: an ω-automaton over the Boolean algebra L with an identified "output" subalgebra $L_P \subset L$ that models the combinational states of the program component. The atoms $S(L_P)$ of the subalgebra L_P are the "output values" of P. By (3), the output values $\mathbf{f}(v)$ represented above (cf. 1), possible at the state v of P, are the elements of $S(L_P)$ that have nonzero conjunction with the transition condition of some state transition leaving v. (The "inputs" to P are the simultaneous collective output values of the various L-processes, i.e., the alphabet $S(L)$. The transitions of P may be independent of any particular outputs, of course.)

If P and Q are L-processes modeling respective program components p and q, the *product* $P \otimes Q$ is an L-process that models the program component formed by taking p and q together. (This sometimes is called the "synchronous product" of p and q, subsuming their "asynchronous" or "interleaving" product [9].) The set of states of $P \otimes Q$ is the Cartesian product of their respective state spaces, and the output subalgebra of the product is the (interior) product of the component output subalgebras.

If $P(v, w)$ is the transition condition for the transition (v, w) from state v to state w of P and $Q(v', w')$ is likewise for Q, then the transition condition for the transition $((v, v'), (w, w'))$ of the product $P \otimes Q$ is $P(v, w) * Q(v', w')$. Some program variables that define a component automaton's transition conditions may come from program components other than the one modeled by the automaton. This allows automata to share conditions on the sequences they accept and reflect coordination among the program components. For example, the call-forwarding execution component modeled by L-process Q refers to the setup component's output variable n that designates the ultimate call destination number. In this way Q coordinates with the setup established by the setup component. The products of their respective transition conditions define the coordinated behavior of the two automata. If $C, D \in L$, then the product of the setup transition condition $P(v, w) = (\mathsf{n} = n_0) * C$ and $Q(v', w') = (\mathsf{n} = n_1) * D$ is nonzero only if $n_0 = n_1$. Thus, the transition condition $(P \otimes Q)((v, v'), (w, w'))$ of the product automaton is nonzero only when the two numbers agree.

The coordination thus defined by automata transition conditions supports the automaton "factoring" mentioned earlier. For any L-process we define its *state transition matrix* to be the matrix over L whose ij-th element is the transition condition for the state transition from the i-th state to the j-th state. By the above definition of process product, the matrix for the product process $P \otimes Q$ is the tensor product of the matrices for P and Q.

In general, if M is the state transition matrix for an L-process that is factored into component L-processes with respective matrices M_i, then

$$M = M_1 \otimes \cdots \otimes M_k. \tag{5}$$

Moreover, if each process is designated by its respective matrix,

$$\mathcal{L}(M) = \mathcal{L}(M_1) \cap \cdots \cap \mathcal{L}(M_k). \tag{6}$$

The intuition for (6) is that each component M_i imposes certain restrictions on the behaviors of the product M, and the behaviors of M are the "simultaneous solutions" for behaviors of the respective components. If M_1 restricts the setup of the call-forwarding number n to numbers in a specified area code and M_2 permits execution only if the designated number n is not busy, then the product pertains to numbers n in the specified area code that are not busy.

Model Checking

We model as respective L-processes the program, its components, and the property to be verified. The program model is the product of its components (5).

It is convenient to use the same symbol to denote a process and its matrix, and henceforth the term *program* will be used to designate both the syntactic computer program and the L-process automaton model into which it gets translated.

Our formal definition of verification of property P for a program M is the automaton language containment check

$$\mathcal{L}(M) \subset \mathcal{L}(P).$$

In words, this says that all behaviors of the program are behaviors "consistent with" (i.e., "of") the property. This is equivalent to checking that

$$\mathcal{L}(M) \cap \mathcal{L}(\hat{P}) = \emptyset$$

where $\hat{P}$ is the "complementary" automaton satisfying $\mathcal{L}(\hat{P}) = S(L)^{\mathbb{N}} \setminus \mathcal{L}(P)$. By (6) verification is transformed into the automaton language emptiness check

$$\mathcal{L}(M \otimes \hat{P}) = \emptyset\,. \tag{7}$$

If M models the C program in Figure 1 according to (2) and P is the automaton that expresses its desired behavior, defined by (4), then verification consists of analyzing the composite system $M \otimes \hat{P}$. Here $\hat{P}$ is P with a complementary acceptance condition that accepts runs for which eventually y = 1. The state of this system is the value of the pair (x, y) , where x is the program variable of Figure 1. The verification algorithm checks whether any state (x, 1) can ever be reached through a succession of state transitions that begins at the initial state (0, 0). The analysis concludes when either a violation of the property is found (i.e., (x, 1) is reached) or all reachable pairs (x, y) have been examined. The latter entails an examination of 2^{32} states, which could be costly were it not for a symbolic technique described below.

The foregoing is an automata-theoretic formulation of model checking. There are other formulations of model checking, expressed in terms of "temporal" logic formula satisfiability [7]. Each of these formulations can be transformed into (7) for some class of automata. In some cases the best way known to perform the check is first to construct an automaton corresponding to the temporal logic formula and then to check (7). See [4], [7], [10] for a way to express the original and widely practiced form of model checking for the logic CTL, without reverting to automata. The automata-theoretic formulation of model checking was developed with a somewhat different perspective from the one presented here by M. Y. Vardi and P. Wolper [12] in the early 1980s and by this author independently around the same time.

Algorithms

A general method to check whether $\mathcal{L}(A) = \emptyset$ for (7) is based on the finite structure of A. $\mathcal{L}(A)$ is nonempty (and the required property fails) if and only if A has an accepting run. This is equivalent to the graph of A's having a cycle "consistent" with the acceptance structure, since A's state space is finite. The path from an initial state to such a cycle may be retraced, and this "error track" provides a step-by-step account of how the property can fail in the program.

One way to check for an accepting run involves an explicit enumeration of the states of A reachable from an initial state. Since the number of states can be exponential in the size of the program description (5), explicitly enumerating the states has severe computational limitations. Fortunately, explicit enumeration is not necessary.

In 1986 R. Bryant made a seminal observation that played a major role in spurring the commercialization of model-checking techniques—a role that resulted in the ACM Kanellakis Prize for Bryant and his colleagues Clarke, Emerson, and McMillan, who showed how to use this beneficially for model checking.

Bryant's observation was that binary decision graphs, long used for planning and programming, could be viewed as automata and thus minimized in a canonical fashion. The minimized structures were dubbed "binary decision diagrams" or BDDs. The idea was to represent the global state as a binary vector and represent a set of states by its characteristic function (having value 1 for each state in the set), encoded symbolically as a Boolean function. The set of reachable states could be represented as a fixed point of the state set transformer that starts with the set of initial states and repeatedly applies the state transition matrix as an operator on the state space, adding the new states reached in one transition, from the set of states reached thus far. Since the set of states is finite, iterating this transformation has a least fixed point—the set of reachable states—in the lattice of state sets. For the Figure 1 program, with respect to checking (7), $A = M \otimes \hat{P}$ is expressed as a Boolean function of x, y (the state variable of $\hat{P}$), input, and added variables x′ and y′ that encode the "next state" of A, as per (1). In terms of a Boolean function,

$$M(\mathsf{x},\ \mathsf{input},\ \mathsf{x}') = (\mathsf{x}' = \mathsf{x} * 2) * (\mathsf{input} = 0) + (\mathsf{x}' = \mathsf{x} * 2 + 1) * (\mathsf{input} = 1) + \cdots$$

and similarly for $\hat{P}$, where + and $*$ denote the respective arithmetic operations and also the Boolean operations "disjunction" and "conjunction" in the Boolean algebra L. As Boolean functions, $A = M * \hat{P}$. If the system starts from the set of initial states defined by the Boolean function I—in this case, the single state that is the solution to $I(\mathsf{x}, \mathsf{y}) = (\mathsf{x} = 0) * (\mathsf{y} = 0) = 1$ (1 being the universal element of L), the states reachable through a single state transition from an initial state are the solutions $I_1(\mathsf{x}', \mathsf{y}')$ to $I(\mathsf{x}, \mathsf{y}) * A(\mathsf{x}, \mathsf{y}, \mathsf{input}, \mathsf{x}', \mathsf{y}') = 1$. With I_1 in place of I, the set of states reachable through two transitions is determined. Since the set of all states is finite, this process may be iterated until no new solutions are found. The cumulative set of solutions defines the set of reachable states. Each iteration is expressed symbolically in terms of the Boolean characteristic functions, reduced to their canonical form and represented as a BDD. A very simple Boolean function can express an arbitrarily large set of states. For example, $x_1 = 0$ defines the set of all global states whose first bit is 0, representing half of the global states, no matter how large the state space. Thus, there is the potential to manipulate very large sets of states. In practice, the ability to compute reachable state sets with 10^{10} states for typical commercial programs is considered fairly trivial, 10^{20} states is routine, and 10^{50} states and higher is not unusual. For the example of Figure 1, the fixed point of 2^{32} states is reached very quickly, in 32 iterations. In each iteration the set of states reached is represented by a computationally simple Boolean function. While the worst-case complexity of performing state reachability symbolically is the same as for explicit enumeration of the states, symbolic search lowered the threshold of acceptability for model checking, leading to its commercialization.

Homomorphic Reduction

A program modeled by an L-process M can be "abstracted" by a "simpler" program (with fewer variables or variables with smaller ranges) that is modeled by an L'-process M'. The relationship between M and M' is given by a Boolean algebra homomorphism $\phi : L' \to L$. If $\phi M'$ is the L-process with transition matrix obtained by applying ϕ elementwise to the transition matrix of M' and $\mathcal{L}(M) \subset \mathcal{L}(\phi M')$, then we say M' is a *reduction* (or "abstraction") of M and M is a *refinement* of M'. If this is the case and, moreover,

$$\mathcal{L}(M') \subset \mathcal{L}(P') \tag{8}$$

for a property defined by the L'-process P', then applying ϕ to both sides gives $\mathcal{L}(\phi M') \subset \mathcal{L}(\phi P')$. So for $P = \phi P'$ it follows that

$$\mathcal{L}(M) \subset \mathcal{L}(P), \tag{9}$$

which means that verifying the reduction verifies the refinement.

One type of reduction, *localization reduction,* described below, is derived by an algorithm. So M' and ϕ are defined algorithmically, and $\mathcal{L}(M) \subset \mathcal{L}(\phi M')$ is guaranteed by construction. Alternatively, a reduction can be "guessed" and then verified as above, where the "guess" consists of a definition of M' and ϕ. As an example of the latter, imagine a component adder used in the context of a program for an integrated circuit that computes the norm of an incoming signal. The correctness of the adder can be established by considering it in isolation from the rest of the circuit. M' in this case could be just the model of the adder, with ϕ mapping the adder to the full circuit with all nonadder variables assigned nondeterministically.

In this example a failure of the isolated adder does not necessarily imply it would fail in the context of the full circuit. If the otherwise-correct adder had errors in case of negative inputs, but negative inputs were impossible in the context of the full circuit, then the failure for negative inputs would not matter, and this "error" in fact may reflect an intentional optimization. In this case one could constrain the inputs to the adder to be positive and prove it correct in that context. This approach would create a "proof obligation" with respect to the remainder of the circuit: to verify that the full circuit does in fact produce only positive inputs to the adder. In discharging such proof obligations, one must beware of circular reasoning: "the adder is good if the rest of the circuit offers it only positive inputs; the rest of the circuit offers only positive inputs if the adder is good" (perhaps a faulty adder gives feedback to the rest of the circuit that can cause the rest of the circuit to offer negative inputs).

In the earlier example of a "complex computation" all nonzero values of the program variable v were abstracted by the value "1", giving another example of a homomorphic reduction. The homomorphism maps v = 1 in the abstract Boolean algebra to the disjunction of all nonzero assignments to v in the refined Boolean algebra.

In order to verify (9), the reduction (8) may be simplified by decomposing the property P into small "subproperties" $P_1, \cdots, P_k$, where

$$\mathcal{L}(P_1 \otimes \cdots \otimes P_k) \subset \mathcal{L}(P). \tag{10}$$

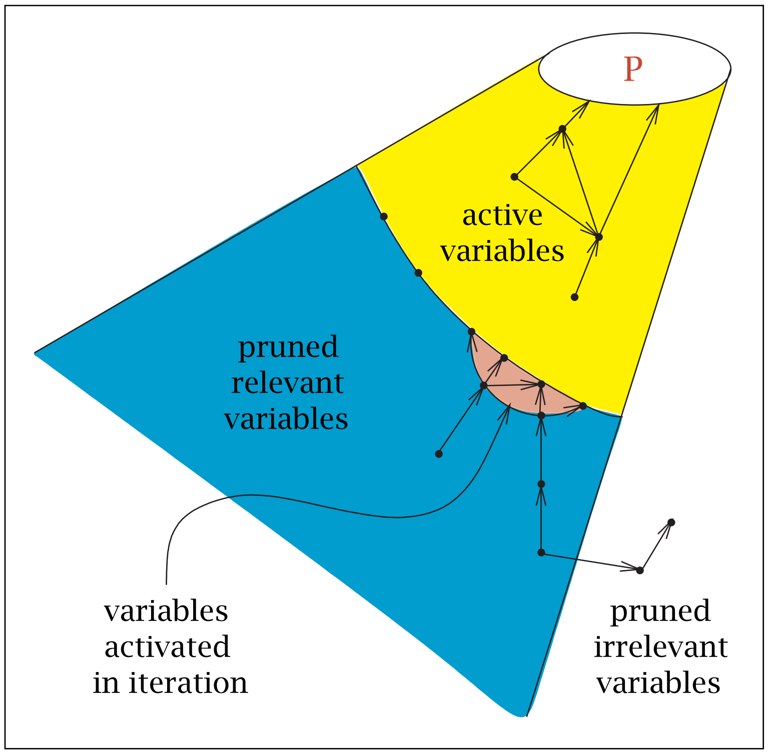

P
active variables
pruned relevant variables
variables activated in iteration
pruned irrelevant variables

Figure 3. Localization Reduction. In the variable dependency graph, a program component close to the automaton P that defines an aspect of the program's required operation is designated *active*. The remaining variables are *pruned*, removing their effect. If the required operation is verified for this reduction, it holds for the full program. If it neither verifies nor falsifies, the active component is augmented, and the check is repeated.

The required check (9) is replaced by

$$\forall i, \quad \mathcal{L}(M) \subset \mathcal{L}(P_i).$$

By (6), this implies (9). Although one check is replaced with k, (9) may entail $O(|M|)$ computations for a model M with $|M|$ sequential states, whereas each of the k component checks can engender a reduction (8), with P_i in place of P, that entails only $O(|M|^{1/k})$ computations. The check (10) can be recursively decomposed into a tree of such checks. At each node of the tree the respective k is small and the P is checked against the product of the P_i's at the successive nodes [9]. If the total number of nodes is $O(\log |M|)$, i.e., is proportional to the number of program components, then the complexity of program verification is proportional to the size of the program. In some cases, when P refers to a "repetitive" structure in M like an array, P can be decomposed algorithmically by restricting it to each successive array element. In general, a decomposition is obtained by a manual "guessing" step, and the guess is verified algorithmically as above.

Reduction and decomposition apply more generally to infinite-state program models. The principles behind them are central to most program verification methodologies that scale tractably with increasing program size.

Design by refinement is an application to verification of a general "top-down" program design methodology that has been around for many years [6], [13]. By evolving a design together with its verification, a development methodology that leads to a tractable verification, as above, may be built into the design process. Program details are added incrementally through a succession of program refinements:

$$\mathcal{L}(M_{i+1}) \subset \mathcal{L}(\phi_i M_i). \tag{11}$$

Here the increasing "level" i indexes progressively more refined program models, leading ultimately to the program with all its details elaborated. Since M_i is a reduction of M_{i+1}, any property verified for M_i holds for M_{i+1}, and on. Properties of the program thus may be verified once and for all at the most abstract level possible, and each level may be verified before the next level is defined.

Based on the composition (5) of each level $M_i = M_{i1} \otimes \cdots \otimes M_{ik_i}$, the check (11) can be decomposed into a set of smaller sufficient checks: for each j,

$$\mathcal{L}(M_{(i+1)h_1} \otimes \cdots \otimes M_{(i+1)h_t}) \subset \mathcal{L}(\phi_i M_{ij}), \tag{12}$$

where the factors $M_{(i+1)h}$ in the successive checks (12) are factors of M_{i+1}.

Localization Reduction

The *variable dependency graph* of a program is the directed graph whose vertices are the program's variables, with an edge (v, w) whenever v appears in the program's assignment expression for w. An automatic way to derive a homomorphic reduction involves a traversal of the variable dependency graph to determine which values of which variables are equivalent, with respect to the property being checked. A variable v is *irrelevant* if it has no directed path to the variables that implement the automaton P that defines the property being checked. In this case, if we transform the program's acceptance conditions to P, the particular values assigned to v can have no bearing on the check (7). Two values of a variable are *equivalent* if the assignment expressions of the relevant variables do not distinguish them. A homomorphism may associate together all equivalent values. Localization reduction is an iterative algorithm that starts with a small "active" program component that is topologically close in the variable dependency graph to P (Figure 3). All other program variables are abstracted with nondeterministic assignments. This renders the values of the variables beyond the boundary of the active variables equivalent, so operationally these variables may be "pruned" out of the model. If the property is thus verified, then by (6) it holds for the full program. On the other hand, if the property fails for this reduction, the algorithm attempts to expand the resulting error track to an error track for the full program. If this succeeds, it means the error in fact reveals a "bug" in the full program. If, however, the expansion fails, it means the error is an artifact of the localization. In this case the active component is augmented by a heuristically small set of topologically contiguous relevant program variables whose assignments are inconsistent with (and thus invalidate) the error track. The verification check is repeated for the expanded program component, and the process is iterated until the algorithm terminates with a verification or a genuine program error.

Conclusion

Program verification can increase the reliability of a program by checking it in ways that may be overlooked in conventional testing. Conventional testing intrinsically comes at the end of the design cycle, whereas program verification may be introduced at the beginning, eliminating bugs earlier than otherwise possible. This can accelerate program development.

A principal type of program verification is model checking, which may be expressed in terms of automata. Although the worst-case time complexity for model checking is for all practical purposes exponential in the size of the program to be verified, model checking often can be accomplished in time proportional to the size of the program, as follows. Define the properties to be checked, decompose each property into subproperties that admit of respective tractable reductions, verify each subproperty on its respective reduction. This approach is "bottom-up", since it begins with the full program.

The alternative "top-down" approach starts with the same property decomposition, but before the program exists. Rank the subproperties according to level of abstraction. For each level define an "abstraction" of the program-to-be. This abstraction corresponds to a reduction in the bottom-up approach. Program details are added incrementally, ending with the fully elaborated program. Each increment is verified to be a refinement of the previous level. A property verified at one level thus remains true at all subsequent levels. Since the program is written and checked incrementally, debugging starts earlier in the program development cycle than with conventional testing, which requires the fully elaborated program. With the top-down approach, the program can be designed so that each required check is tractable.

Reductions can be derived algorithmically in the bottom-up approach. In either approach a prospective reduction may be verified to be an actual reduction by checking the refined program against a homomorphic image of the reduced program.

For both the bottom-up and top-down approaches, property decomposition is a fundamental step. In special cases, when a property refers to a repetitive structure, its decomposition can be derived algorithmically. In general, decomposition can be verified—but not derived—by a tractable algorithm. In fact, it is not tractable to determine whether a "good" decomposition—one that gives rise to tractable bottom-up verification—exists. Finding useful heuristics for decomposition is a foremost open problem in model checking.

Acknowledgments

I thank Ron Hardin, Tony Knapp, Peri Kurshan, Jeff Lagarias, Henry Landau, Susan Landau, Ken McMillan, Kedar Namjoshi, Andrew Odlyzko, Kavita Ravi, Marty Reiman, Chuck Vinsonhaler, Alan Weiss, Margaret Wright, and Mihalis Yannakakis for their very helpful comments on drafts of this article.

References

[1] J. Barwise, Mathematical proofs of computer system correctness, *Notices Amer. Math. Soc.* **36** (1989), 844–851.

[2] W. W. Bledsoe and D. W. Loveland (eds.), *Automated Theorem Proving: After 25 Years*, Contemporary Math., vol. 29, Amer. Math. Soc., Providence, RI, 1984; especially R. S. Boyer and J. S. Moore, Proof-checking, theorem-proving and program verification, pp. 119–132.

[3] E. M. Clarke and E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, Lecture Notes in Comp. Sci., vol. 131, Springer-Verlag, Berlin and New York, 1981.

[4] E. M. Clarke Jr., O. Grumberg, and D. Peled, *Model Checking*, MIT Press, Cambridge, MA, 1999.

[5] R. DeMillo, R. Lipton, and A. Perlis, Social processes and proofs of theorems and programs, *Comm. ACM* **22** (1979), 271–280.

[6] E. W. Dijkstra, Hierarchical ordering of sequential processes, *Acta Inform.* **1** (1971), 115–138.

[7] E. A. Emerson, Temporal and modal logic, *Handbook of Theoret. Comp. Sci.*, vol. B, Elsevier, Amsterdam, 1990, pp. 995–1072.

[8] P. Halmos, *Lectures on Boolean Algebras*, Springer-Verlag, Berlin and New York, 1974.

[9] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes—The Automata-Theoretic Approach*, Princeton University Press, Princeton, NJ, 1994.

[10] K. L. McMillan, *Symbolic Model Checking*, Kluwer, Dordrecht, 1993.

[11] S. Schroeder, Turning to formal verification, *Integrated System Design Magazine* (Sept. 1997), 1–5.

[12] M. Y. Vardi and P. Wolper, *An Automata-Theoretic Approach to Automatic Program Verification*, Proc. (1st) IEEE Symp. Logic in Comput. Sci., Boston, 1986, pp. 322-331.

[13] N. Wirth, Program development by stepwise refinement, *Comm. ACM* **14** (1971), pp. 221–227.