SOFTWARE FOR ANALYTICAL DEVELOPMENT OF COMMUNICATIONS PROTOCOLS

Zvi Har'El and Robert P. Kurshan

__Zvi Har'El__ is a former member of technical staff in the Mathematics of Communication and Computer Systems Department at AT&T Bell Laboratories, Murray Hill, New Jersey, to which he remains a consultant working on the COSPAN software project. Currently, he is a Senior Lecturer in the Department of Mathematics at the Technion—Israel Institute of Technology, Haifa. He holds B.Sc. and D.Sc. degrees in mathematics from the Technion. __Robert P. Kurshan__ is a member of technical staff in the Mathematics of Communication and Computer Systems Department at AT&T Bell Laboratories, Murray Hill. His current interests include hardware and software development through hierarchical formal verification. Mr. Kurshan joined the company in 1969 after earning a Ph.D. in mathematics from the University of Washington, Seattle.

We describe a way to develop and implement communications protocols so they are logically sound and meet stated requirements. Our methodology employs a software system called the *coordination-specification analyzer* (COSPAN) to facilitate logical testing (in contrast to simulation or system execution testing). Logical testing of a communications protocol is carried out on a succession of models of the protocol. Starting with a high-level model (e.g., a formal abstraction of a protocol standard), successively more refined (detailed) models are created. This succession ends with a low-level model which is in fact the code that runs the ultimate implementation of the protocol. Tests of successive models are defined not by test vectors, but by user-defined behavioral requirements appropriate to the given level of abstraction. Testing a high-level design permits early detection and correction of design errors. Successive refinement is carried out in a fashion that guarantees properties proved at one level of abstraction hold in all successive levels of abstraction. We recount the experience of an application of this methodology, employing COSPAN, to develop (analyze and implement in software) a new session protocol at an interface of an AT&T product called the Trunk Operations Provisioning Administration System (TOPAS).

Introduction

Speaking very loosely, the overall objective of a communications protocol development project could be stated as follows: *To develop a reliable product that is fast (in absolute days), efficient in its use of allocated staff, supportable, well-documented, robustly designed*

Panel 1. Terms and Acronyms in This Paper

AUTO	autonomous session protocol, a non-standard protocol within TOPAS
CCITT	International Telegraph and Telephone Consultative Committee
COSPAN	coordination-specification analyzer
CPU	central processing unit
FTAM	file transfer access and management
I/O	input/output
Lv1	Level 1, numerical level applied to a high-level protocol specification developed using COSPAN
Lv2	Level 2, buffering and channel interface level of AUTO derived from Lv1 above
Lv3	Level 3, AUTO's implementation level
PC	personal computer
S/R	automaton-language used for system development
s/r	selection/resolution coordination model
TOPAS	Trunk Operations Provisioning Administration System
URP	universal receiver protocol
VLSI	very-large-scale integration

(i.e., insensitive to changes in environment behavior), and portable among various settings. In pursuit of these objectives, one may apply a variety of "techniques" with familiar names: *Rapid Prototyping, Design For Testability, Modularity, Hierarchical Development, Structured Team Management*, and *Design For Reusability.*

While these development goals and the techniques to obtain them mean different things to different people, system designers and developers are increasingly aware of the value of *formal* techniques to define and achieve such goals. Not only is the international community working on standards for protocol specification and testing based upon formal techniques, but many companies have developed their own formal techniques, both for particular projects and for general use. Within the scope of a formal technique, many of these terms assume specific, focused meanings, concomitant with good predictability and reliability of the methodology.

The advantage of definition by itself, however, is of little more than academic value unless formalism is applied to some specific advantage. Perhaps the most obvious application of formalism involves product reliability. Once a protocol design and requirements have been formalized, it becomes theoretically possible to determine (in a mathematical sense) whether a particular implementation meets these requirements. This is known as *formal verification.* The value of a particular formal verification framework may be measured by the scope or generality of the requirements for which an implementation may be tested.

For example, a framework where only "safety" properties may be verified (properties of the form "such-and-such bad event cannot occur"), may be inadequate for analyzing communications protocols, where a paramount concern is, for instance, that once a message is transmitted, it is eventually received (a non-safety property).

Even if the immediate objective is only to "certify" that a given implementation meets stated standard requirements, presumably one needs to verify as well that the stated standard requirements are enough to guarantee the overall good behavior of the protocol. Unfortunately, protocol behavioral requirements extend beyond those associated with certification. To ensure the proper behavior of a particular implementation, extensive behavioral verification must be conducted.

It is now generally accepted that simulation or system execution is far from adequate to ensure the proper behavior of an implementation. Therefore, formal verification should have the power to draw firm conclusions about the general behavior of a system under all possible situations.

Because protocol requirements address broad properties of a protocol, formal verification is most easily (and most often) applied not to the implementation, but to a high-level model or abstraction of the implementation. It is assumed implicitly that an implementation will be true to the verified high-level model. There is a similar relationship for "certification" (testing an implementation against a standard). In this case the standard is a high-level model; ideally, certification would constitute a proof that the implementation is a correct realization of the standard, or equivalently, that the standard is a correct abstraction of the implementation. Thus, for certification of an implementation or verification of a high-level model to guarantee the correctness of the implementation, there must exist a formal association or transformation from the high-level model or standard to the low-level implementation. Let us examine this more closely.

When a protocol is defined by a high-level model that has been verified to satisfy certain requirements, it is common to say *the protocol has been verified.* The next step is to implement the verified protocol model. If the implemented protocol then fails on account of an implementation decision (i.e., a construct in the implementation not described precisely in the model) or on account of a misinterpretation of the model, it is common to attribute the failure not to the "protocol" (i.e., the protocol model) but to the implementation. Given the preponderance of failures caused in practice by such "implementation decisions" and translations of the model, it would seem to make little difference, however, where the blame is placed. The best methodology for formally verifying a high-level model has limited value if there is no formal procedure to derive a faithful implementation from it. Often there is not even a formal basis upon which to decide if an implementation implements a model. This can be a particular problem when it masks the fact that *no* implementation of the protocol, precisely as modeled, is possible in a given environment. Such a difficulty arises—and not all that uncommonly—when a protocol model contains implicit assumptions about environment interfaces which are not met by the given environment. If so, *any* "implementation" of the protocol model in the given environment is necessarily untrue to the model, and thus properties verified in the model may not hold for the implementation. Likewise, if a certification scheme involves tracking an implementation with a high-level standard, in order for this to give real information about the implementation, a formal connection is needed between the standard and the implementation. If there is a formal connection—such as an association of states and transitions—then proper certification should demonstrate that *every* transition of the implementation corresponds to the associated transition in the standard.

Even in the absence of a formal transformation from protocol model to implementation, formal analysis of the model can reveal faults in the protocol concept. However, in the absence of such faults, it may be deceptive to refer to an implementation as "verified" if there is no formal, testable relationship between the verified model and its implementation. Likewise, a certification procedure that tracks an implementation with a standard can be very useful to the extent that it uncovers discrepancies between the standard and the implementation. Without such discrepancies, however, it may be deceptive to refer to an implementation as "certified" if there is no formal, testable relationship between the standard and the implementation, or if only a few transitions of the implementation have been tested.

A simple way to define a formal relationship between a high-level model or standard and an implementation is to associate a state in the model or standard with a set of states of the implementation. Such an association, for example, may require correspondence between the *receiver-ready* state of the high-level model and the set of implementation states for which a certain state machine *component* of the implementation is in its receiver-ready state. However, since the set of implementation states for which this state machine component is in its receiver-ready state is determined by all the possible respective values of pointers, buffers, counters and so on which may occur with it, this set of states probably is very large.

Suppose that, according to the high-level model or standard, the correct transition from receiver-ready is to the state *transmit*. It may be that for certain implementation states (i.e., for certain values of pointers, buffers and so on), the implementation tracks the model or standard, while for others it does not. To certify truly that a high-level model or standard abstracts an implementation, one must demonstrate this not simply for a single implementation state and transition that corresponds to a respective high-level state and transition, but rather for *every* low-level state and transition. Probably, in lieu of symbolic verification methods, the best approach is the standard simulation routine that runs the implementation alongside the high-level model or standard for as long as feasible (inevitably crossing high-level transitions many times). Nor is it accurate to conclude that if a high-level transition—e.g., from receiver-ready to transmit—correctly tracks a single low-level transition (for a certain value of buffers, pointers, etc.), then in greatest likelihood, the receiver-ready to transmit transition would correctly track *all* other low-level transitions between the corresponding states (i.e., for all other possible respective values of buffers, pointers, etc.). Indeed, it is well-known that the greatest weaknesses of an implementation arise at the "boundaries" of operation (buffer empty, buffer full, etc.), and that these boundaries can be very long and complex.

Since it is rarely feasible to address directly all possible transitions of an implementation (i.e., to address all possible values of buffers, pointers, etc.), one must seek alternatives by which to conclude that an implementation is faithful to its high-level abstraction.

All this may sound very complicated, and indeed in purely numerical terms it certainly is. While a high-level model or standard may have as few as 50 or 500 states, an implementation typically has so many states that the numbers can be appreciated only by analogy. The implementation of the session protocol we describe in this paper is about average in size. Given all the possible combined values of its pointers, buffers and state machine controllers, its state-space contains an estimated 10^{10^4} reachable states.

To understand this number, one may think of it this way: to determine if our implementation tracks a high-level model, we could, in theory, use a brute-force state-transition tracking algorithm. Suppose this algorithm were perfectly parallelizable among every human being on earth, each equipped with a Cray computer. Even so, the job could not be completed before the sun burns out.

Methods to manage the overwhelming complexity caused by the formal methods themselves must be supported. The classical method—***test as much as you can and then hope for the best***—now generally is agreed to be inadequate. More powerful methods must be found. Using mathematics, we are able to reduce apparently intractable tests such as the one just described, to a relatively simple test with the provable property that the outcome of the simple test reflects conclusively upon the outcome of the apparently intractable test (if it had been performed).

We present a high-level to low-level transformation in the form of a formal top-down development procedure based upon successive refinement. Starting with a high-level model (e.g., a formal abstraction of a protocol standard), successively more detailed models are created through successive refinement, in a way that guarantees that properties verified at one level of abstraction hold in all successive levels of abstraction. This succession ends with a low-level model consisting of code which runs the ultimate implementation of the protocol.

We may contrast this with conventional development. Conventional development procedures add functionality in the course of development. As functionality is added, so are possible behaviors of the system. This precludes the possibility of a property proved early in the development cycle can be assured at the end of development. In the development methodology we utilize, the implementation has *fewer* behaviors than the high-level design. This is what guarantees the inheritance of properties proved early in the design cycle.

Formal verification of quite general (in fact, arbitrary finitary-automaton-definable or *ω-regular*) user-specified requirements, appropriate to the given level of abstraction, is facilitated by reduction techniques. For each system requirement to be verified, we derive a reduction of the system *relative to that requirement.* The scope of such ω-regular requirements includes not only "safety" properties but also "eventuality" properties such as "the message eventually will be delivered."

The high-level model at the top of the development hierarchy constitutes a prototype of the system being developed. Because this model may be as abstract (and hence simple) as one likes, it may be defined quite rapidly, providing a *rapid prototype* of the system under development.

Partitioning the formal verification procedure according to the levels of abstraction defined by the development hierarchy constitutes *design for testability* that facilitates verification.

The basis of the hierarchical development is a *modularity* that permits successive module-by-module refinement rather than the hopeless task of globally refining a monolithic system. The modules may be designed to be small, simple components, so refinements can be easily designed and verified. Modular design, the partition imposed by the development hierarchy and separate verification of numerous requirements (as dictated by the reduction techniques), all impose a natural separation of the development project into semi-independent tasks that can be performed by a development team according to a schedule imposed by the development hierarchy.

Another by-product of the development hierarchy is a form of *reusability*, where the same higher-level model may be developed into a variety of lower-level models. Conventional attempts to design reusable code often are frustrated by the need to alter substantial parts of the code for different interfaces. This altering can have unpredictable consequences in the reused portions. The form of reusability proposed here replaces interface redesign with *refinement*, thus preserving properties of the higher-level reused model.

Formal verification buys *reliability*. Rapid prototyping concomitant with hierarchical verification allows error-detection very early in the design and development process, eliminating the much greater time and cost of redesign *after* development is complete. This advantage may decrease development time by an order of magnitude (see below) over conventional methods. The development methodology naturally partitions development into parallel and generally independent tasks. Thus, the combined efforts of the development staff interfere less—and hence are more efficient—than in development projects where a change in one part of the project inevitably forces changes in many other parts.

The development hierarchy and system modularity also ease the burden of system support. That is, new features may be introduced at the appropriate level, verified and refined into the implementation without disturbing previously defined and verified system components. Because the development hierarchy affords views of the system at a variety of levels of detail, the system design is almost *self-documenting.*

Because low-level system interface details are introduced only at a low level in the development hierarchy, the resulting design tends to be robustly constructed relative to changes in the environment behavior definition. Likewise, the higher level system design tends to be *portable*, requiring only redefinition of the lowest (or lower) levels to implement the system in various settings.

In summary, formalism offers the potentials of formal verification, a formal relationship between a high-level model and its implementation, and management of the apparently intractable complexity inherent in the

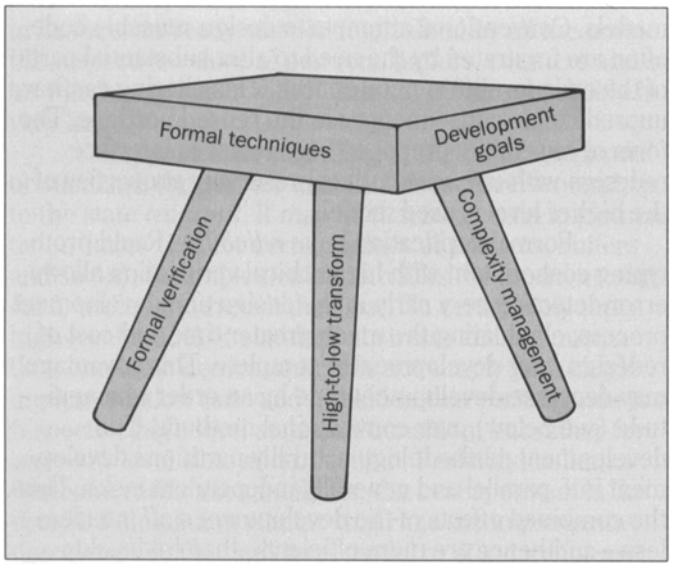

Figure 1. The three legs of formal techniques: without any one leg, the overall objectives would be substantially compromised.

formalism itself. To use formal techniques to support the development goals stated at the start, each of these three potentials is required by the others; like the legs of a stool (Figure 1), each is indispensable. Taken together, they can support the development goals through a formalization of familiar techniques.

The COSPAN Software System

In this section we outline the concept and use of COSPAN, a general-purpose software tool for *coordination-specification analysis*. Typical applications of COSPAN include software development and hardware development for implementing control-intensive structures, prominently communications protocols. COSPAN has been used in the commercial development of both software and hardware to implement communications protocols. We will describe one such project here.

To develop software or hardware to implement a communications protocol, a target system implementation is created through a formal top-down development procedure based upon successive refinement. Starting at a very abstract level of definition, a system is repeatedly redefined, adding more detail at each level of definition. The target software or hardware is *automatically* generated from the final, most detailed level of definition. For software, this means production-quality C-language code; for hardware, it means a circuit net description and C-code for input to a silicon microprocessor layout tool. In either case, a variety of optimization steps are performed during development.

One of COSPAN's major features is a set of routines for logically analyzing the developing system at each level of definition. Using these routines permits error detection from the earliest developmental stages onward.

Logical analysis consists of *symbolic* system tests (*not* simulation or system execution) for quite general user-defined behavioral requirements. The analysis constitutes a mathematical proof (or disproof) of fulfillment of the stated requirements. For example, a typical high-level behavioral requirement for a communications protocol is that all messages generated at the *sending* end are eventually received by the *receiving* end, in the order sent and without duplication.

Generally, the potential scope of our formal verification is quite broad, encompassing all ω-regular assertions, as explained below. The top-down development procedure guarantees that requirements satisfied at one developmental level will remain satisfied at all subsequent levels. This procedure constitutes a formal transformation from the high level to the low level. Central to the tractability of COSPAN's analysis algorithms are formal reduction procedures to cope with the typically enormous state-spaces associated with virtually all production-oriented coordination problems. These procedures consist of simplifying the system *relative to the property being verified.*

Reduction. COSPAN is based upon an automata-theoretic discrete-event model (see below) that incorporates arbitrary delays between events. This makes it especially suited to analyze asynchronous coordination. The system under development is defined modularly, in terms of coordinating state machines, while system requirements are defined in terms of automata. Both the system and its requirements are defined in terms of the same syntax, that of an automaton-language called S/R,[1] whose semantics is given by the s/r model described below. Mathematically, analysis consists of testing whether the formal language associated with the system under development is contained in the language associated with each of several user-defined ω-regular requirements, called *tasks*. (The ω-regular languages are like the regular languages, except their members are sequences rather than strings. This, and other formalisms mentioned in this section, will be described more fully in the next section.)

Each language containment test is performed in time which is linear in the size of the system and the task. Notwithstanding this linear-time test, typically the system—and sometimes the task as well—are intractably large. To reduce the complexity of this test, we construct a special map, called a ***homomorphism***, for each task.

Each homomorphism reduces the analysis of the intractably large system and task to analysis of a smaller (computationally tractable) system and task. Consequently, if the smaller system performs the smaller task, the original system performs the original task. Intuitively, a homomorphism "hides" portions of the system that are irrelevant to the performance of the particular task. Fortunately, constructing such a homomorphism does not require constructing the intractably large system itself. Rather, a system homomorphism may be constructed *implicitly* through explicit homomorphisms defined on system components, which may be designed to be quite small. Proof of the constructibility of a system homomorphism from the component homomorphisms is founded in the Boolean algebra of the enabling predicates for the transitions of the underlying state machines and automata.

A homomorphism of the entire system can be constructed from the component homomorphisms as long as their action is consistent on this Boolean algebra. The Boolean algebra contains a logical description—in terms of state machine outputs—of the conditions under which each component changes state, and may be visualized as a unifying sort of "glue" through which the modular system components are connected.

Homomorphisms are used to define the successive refinements of a system, as well. As with reduction, this homomorphism guarantees that any task performed by the more abstract system also will be performed by the more detailed system. Both in the case of reduction and refinement, a homomorphism is defined by the user and the validity of this definition is verified by COSPAN.

When used in the course of top-down development, COSPAN is a rapid-prototyping tool, providing a theoretically seamless interface between the abstract model or standard and the target implementation. COSPAN supports facilities for documentation, conformance testing and software maintenance, as well as facilities for libraries of abstract data types and reusable pretested components. It supports debugging tools for pinpointing the source of task-performance failures, and facilitates stochastic analysis to test timing and statistical performance.

The state machine architecture defined by programs written in the S/R language is inherently modular. This, together with the emphasis on early analysis, lends itself well to team development. The top-down design methodology introduces a new form of reusability: the same high-level model, tested and true, may be refined into several implementations.

COSPAN programs are written in S/R, an automaton-language that can be used in each coordination problem to build a procedural base supporting declarative structures. In such utilizations, S/R creates ad hoc abstract data types that define special-purpose dataflow languages. This feature makes S/R particularly suitable for developing distributed or state-machine-based environments (such as communications protocols and highly parallel hardware) where structure and behavior naturally are understood in terms of data flow.

The Underlying Semantic Model

The use of state machines to model non-terminating processes—such as communications protocols and complex integrated hardware systems—is conceptually attractive because it affords a well-understood mathematical model with an established literature. However, it has long been recognized that a serious limitation of the state machine model in this context is the size of its state-space, which grows exponentially with the number of coordinating components in the protocol or system. Because most protocols or hardware systems have many coordinating components, the pure state machine model has been all but dismissed from serious consideration for formal verification of these systems. The enormous size of the ensuing state-space has been thought to render its analysis intractable.

However, in using COSPAN to facilitate a methodology based upon homomorphic reduction and refinement, we have seen that this is not necessarily so. By exploiting the symmetries and modularity commonly designed into a large coordinating system, we can test an apparently intractable state space for a regular-language property or "task" by examining a smaller associated state-space. The smaller state-space, which is a reduction relative to the given task, has the property that the original system performs the given task if the reduced system performs a reduced task.

Checking the task performance of the reduced system amounts to testing whether the ω-regular language associated with the reduced system is contained in the language that defines the reduced task. For a new class of automata well-suited to defining properties of coordinating systems,[2] testing can be performed in time that is linear in the number of edges of the automaton defining each reduced language. (For Büchi automata, commonly associated with such testing, testing language containment is essentially intractable: mathematically, it is P-SPACE complete.[3]) All ω-regular languages may be expressed by this new class of automata.

Finite state automata which accept sequences (rather than strings) define the ω-regular languages. This class of automata is established as a model in logic, topology, game theory and computer science.[3–9] In computer science, such automata are used to model non-terminating processes such as communications protocols and complex integrated hardware systems.[10–17]

In the context of modeling non-terminating processes, conceptually one considers a state machine M and an automaton T. The state machine M models the process under study, together with its environment (the protocol or hardware system, for example), whereas the automaton T models the requirement or task which the process is to perform. Although the state-space of M may be too large to permit M to be constructed explicitly, M may be defined implicitly in terms of a tensor product of components, as described below. Likewise, the task-defining automaton T (defining a property for which M is to be tested) also may be defined implicitly, in terms of components.

To determine whether the process performs the specified task, we check if the language $\mathbf{L}(M)$ defined by M is contained in the language $\mathbf{L}(T)$ defined by T.

The precise conditions[2] under which the test $\mathbf{L}(M) \subset \mathbf{L}(T)$ may be replaced by the simpler test $\mathbf{L}(M') \subset \mathbf{L}(T')$, for smaller M' and T' are beyond the scope of this paper. We simply point out that M' and T' are derived from M and T through *co-linear automaton homomorphisms*, maps that are graph homomorphisms, preserve the transition structure of the respective state machines and automata, and agree on a Boolean algebra associated with the underlying alphabet that defines the enabling transition predicates. These homomorphisms, described more fully below, may be constructed implicitly from explicit homomorphisms on components defining each of M and T, thus avoiding construction of the product spaces M and T themselves. Component homomorphisms are user-defined, then machine-verified. While there is no guarantee that a particular intractably large problem can be thus rendered tractable, experience has demonstrated the utility of this approach (see *The Story of One Project* below).

The intuition behind such a reduction is quite simple. The reduction simply formalizes the notion that, to test a given system for a stated property, many system actions may be suppressed because they are irrelevant to the property. For example, if the property concerns a *sender* working in conjunction with an *interrupt-handler*, it may be possible to abstract the actions of the interrupt-handler if the property being tested has nothing to do with the handling of interrupts. (Note that such a reduction has little to do with state-space minimization, since the reduced system here is not equivalent to the original system.)

In top-down development, the relationship of one level of abstraction to the next is defined by a homomorphism as well. This homomorphism is semantically identical to the homomorphism just described. In fact, there is an intimate relationship between reduction and development that now can be clarified. If the state machine M above is taken to be the lower-level refinement of the more abstract M', then the assertion that a task proved performed by M' is necessarily performed by M is simply the implication $\mathbf{L}(M') \subset \mathbf{L}(T') \Rightarrow \mathbf{L}(M) \subset \mathbf{L}(T)$. Here, T' is a task for the more abstract system M' and T is the corresponding task for the refined system M. To verify that the lower-level system M is a refinement of the more abstract system M', a homomorphism $M \rightarrow M'$ is user-defined and machine-verified. The corresponding task T is defined implicitly by T' and this homomorphism.

In the course of top-down development, nondeterminism that permits a degree of abstraction at higher levels, is resolved at lower levels into deterministic actions determined by the lower level mechanisms. One result of this is that lower levels exhibit fewer behaviors than higher levels. (This is used to show that properties proved at higher levels are retained by lower levels.)

The semantic link between state machines, automata and protocols is the *s/r* (selection/resolution) model of coordination.[9] The s/r model is a discrete-event model of concurrently coordinating entities, with the semantics of automata on ω-languages. In this model, a system is defined by a collection of Moore-like[18,19] state machines. Transitions in each component machine are enabled by conditions on the outputs of the various component machines. These enabling conditions, or *transition predicates*, are represented by Boolean functions of the outputs of the component machines. Under the Boolean operations of AND, OR and NEGATION, these transition predicates generate a Boolean algebra.[20]

Through this Boolean algebra, one may readily compute the single *product* machine which represents a system of k component machines operating concurrently: the state-space of the product machine is the Cartesian product of the respective state-spaces of the component machines. The transition predicate associated with the product machine transition

$$(v_1, v_2, ..., v_k) \rightarrow (w_1, w_2, ..., w_k)$$

is the k-fold (Boolean) AND of the k transition predicates associated with the component machine transitions $v_i \rightarrow w_i$ for $i=1, 2, ..., k$. Indeed, each component machine M_i $(i=1, ..., k)$ may be represented as a matrix over the Boolean algebra; the *vw*-th component of M_i, $M_i(v, w)$, is the Boolean predicate which defines the enabling predicate for the transition of M_i from its component state v to its component state w. The *product* M of $M_1, ..., M_k$ then is represented by the tensor product of matrices $M = M_1 \otimes \cdots \otimes M_k$ defined by

$$M(v, w) = M_1(v_1, w_1) * \cdots * M_k(v_k, w_k)$$

where $v = (v_1, ..., v_k)$, $w = (w_1, ..., w_k)$ and $*$ is the

Boolean AND. For any machine M over this Boolean algebra (for example, the product machine, or any component machine), the *language* $\mathbf{L}(M)$ of the machine is defined as the set of (infinite) sequences of global machine outputs which are consistent with the transition structure of the given machine. Thus, through the Boolean algebra, one may view $\mathbf{L}(M)$ as the set of all possible system behaviors supported by M.

A homomorphism $M \rightarrow M'$ maps states of M into states of M' (in a many-one fashion), transition predicates in M to (abstracted) transition predicates in M', and likewise maps T to T', in such a way that every behavior (sequence of outputs) of M not accepted by T, maps onto a corresponding behavior of M' that is not accepted by T'. A particularly trivial form of homomorphism is state-space minimization. That is, a state machine sometimes may be replaced by another state machine with fewer states, but the same input-output behavior. More significant, however, are homomorphisms involving abstractions of inputs and outputs; these typically lead to much greater reductions in state-space size than state-space minimization alone, and unlike minimization, may be defined on non-deterministic transition structures. (Minimization must be performed on a deterministic structure; determinizing a non-deterministic structure for the purpose of state minimization can actually add many more states than minimization removes.[18]) For example, suppose that for two machine outputs, each with range 1, ..., N, the correctness of a particular task is independent of the respective output values, as long as the two outputs have the same parity. Then the N^2 possible values of the output pair may be abstracted to 2 values: "same" and "opposite." If each output is associated with a state with the value of that output, then N^2 states thus are homomorphically reduced to 2 states. Since the original outputs distinguish states, state-space minimization alone would provide no reduction. If the original transition structure were non-deterministic, then determinizing it in order to perform (a futile) state minimization could actually result in a "reduced" state-space of as many as 2^{N^2} states.

Formally, a homomorphism is defined as follows. Let $M(v,w)$ be the condition for M to make a transition from state v to state w, and let $M'(v', w')$ be the analog for M'. Furthermore, let Φ_s be a (many-to-one) mapping from the states of M to the states of M', and let Φ_t be a mapping from transition predicates of M to transition predicates of M'. (As a technical detail, to ensure the consistency of Φ_t, we must require Φ_t to be derived from a Boolean algebra homomorphism from the Boolean algebra of the simpler machine M' to the Boolean algebra of the more complex system M.[2]) We say that $\Phi = (\Phi_s, \Phi_t)$ is a *homomorphism* from M to M' if, for all v, w,

$$\Phi_t M(v, w) \leq M'(\Phi_s v, \Phi_s w) ,$$

where $\leq$ is the inherent partial order in the Boolean algebra, that is, if whenever M can make a transition from v to w, then M' can make a corresponding transition from $\Phi_s v$ to $\Phi_s w$.

Applications of COSPAN

COSPAN has been used to analyze high-level models of several communications protocols, including the X.25 packet switching link layer protocol and the file transfer and management protocol (FTAM) of the International Telegraph and Telephone Consultative Committee (CCITT), and AT&T's Datakit® universal receiver protocol (URP) level C. In such applications it has proved useful as a debugging tool. However, in these applications it could not guarantee correct operation of the protocol under study, since COSPAN was used only to analyze the high-level models, without relation to their implementations. Thus, although errors found by COSPAN were used as a guide to fix the specification and check the implementation, there was no high-to-low level transformation, and thus no verification of the implementation.

However, COSPAN presently is being used to develop a custom VLSI chip to implement a packet layer protocol controller. In the past, it has been used to develop into software two session protocols for an interface to an AT&T product called the Trunk Operations Provisioning Administration System (TOPAS). We describe the experience of that software project in the following section.

The Story of One Project

This section describes a case history of one COSPAN application that may be presumed to be typical. A development group in AT&T Bell Laboratories decided to use COSPAN to develop and implement a non-standard session protocol, the autonomous message protocol (AUTO) at an interface with TOPAS.

A COSPAN-generated implementation, derived through three developmental levels, was produced in about 10 percent of the budgeted time. If the time budget accurately represents the cost of conventional development (something we suspect, but cannot know for sure), this is quite remarkable. The automatically generated code was deemed as good as—and possibly much better than—what could have been written by hand. As expected, no errors were found during system test. The total absence of errors in system test represented a marked departure from the experience of conventional development. Furthermore, this absence of errors was thought to represent a considerable saving in subsequent time and effort, for no subsequent revisions would be needed after system test.

During development, at least 50 significant logical errors were uncovered and corrected by COSPAN, including system design errors that required some system redesign. Two of the system design errors were judged as the kind usually discovered only in subsequent issues of a system. If this is taken into account, the saving in implementation costs may be higher than the 90 percent saving in development time. There also was the benefit of releasing a more reliable product in the first issue.

System Design. Though basically a classical "go-back-*n*" protocol[21], AUTO involves an additional twist of a "journal" into which messages are sometimes buffered and then delivered out of order (with lower priority) relative to non-journaled messages.

Before COSPAN was applied to this project, high-level system design had been completed, and system requirements were described in official baselined documents. Six additional months of two to three programmers' time had been allocated to the development of AUTO, which was anticipated to require approximately 10,000 lines of C-code. After this, the software was to be officially "handed off" to system test, allowing time for a few more rounds to fix defects. The final version of the software—the first release—was to be ready after nine months.

The S/R Specification. Working from the baselined documents and one to two days of face-to-face discussion with the design engineers, a single programmer completed a high-level S/R language specification of the protocol in about 10 days. This specification, dubbed Lv1 (Level 1) contained all the logic of the protocol, but abstracted buffering, channels and messages. Figure 2 illustrates the dataflow architecture of AUTO. AUTO Lv1 was tested for the performance of two tasks: *del* (every message which is sent is ultimately delivered) and *seq* (no message is ever delivered twice with different sequence numbers).

Testing. Testing and revision of AUTO Lv1 required another six to eight weeks of work for one programmer. During testing, about 100 errors were uncovered in the specification (i.e., AUTO Lv1). Of these, about 50 percent were minor (typographical) errors, requiring a change in only a couple of lines of the S/R specification, while about 25 percent required rewriting significant portions of Lv1. Certainly, the largest number of these errors were the fault of the implementation. However, at least three of them were significant design errors, reflecting self-contradictory system design that would be impossible to implement correctly. These system-design errors required redesign by the design engineers—and a reapproval phase beyond the six to

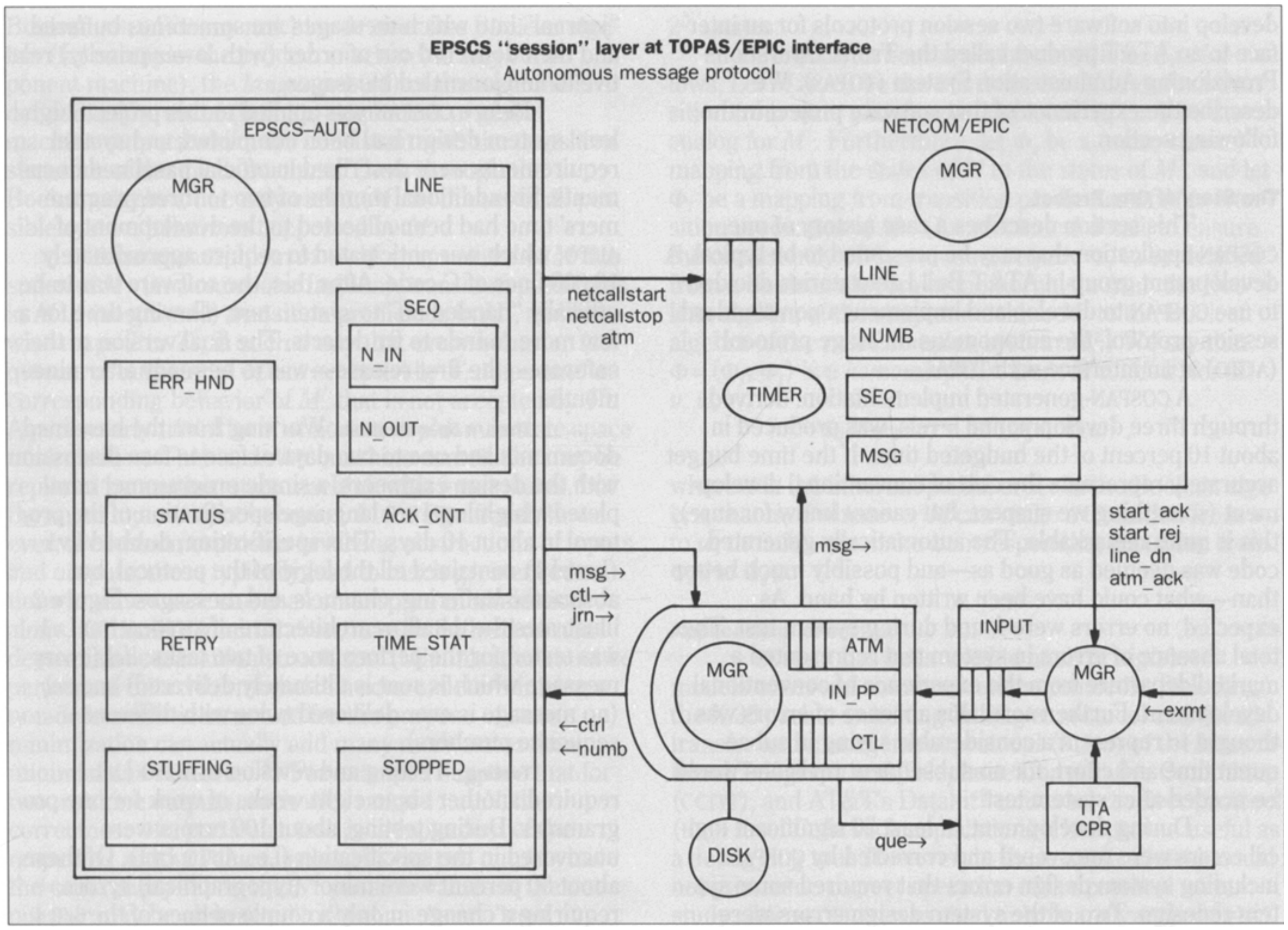

Figure 2. Level 1 (most abstract) data flow architecture of AUTO protocol. Each box is specified as a state machine. The arcs indicate the flow of tokens at this level abstractions of data. The labels on arcs are names of the tokens. The two boxes EPSCS-AUTO and IN_PP designate the protocol to be implemented. The other three—NETCOM/EPIC, INPUT, and DISK—model structures of the environment.

eight weeks—before the COSPAN-driven development of AUTO could proceed.

Of the six to eight weeks taken to debug Lv1, about 80 percent went to successively rewriting the specification, while the rest went to setting up the testing itself. Testing in COSPAN comprises three steps: *reduction*, *analysis* and *fault-location*.

Reduction. Formal reduction of the specification circumvents what otherwise would be an intractably large state-space (e.g., 10^{10^4} states). With some reductions, an error found in the reduction reflects an error in the specification, but not necessarily conversely. Such reductions, since they are easily generated, typically are used at the beginning of testing when many errors are expected. (COSPAN automatically finds and reports any errors in a given reduction, as described in the *Analysis* section below.)

Once no errors are found, a reduction must be found where no error in the reduction implies no error in the specification. The latter reductions are needed to prove the system is error-free, and are harder to find (but in all cases could be found). Once designed, they require one or more runs of COSPAN to prove they are correct reductions.

A typical example of the latter type of reduction amounts to replacing a circular queue with a push-down queue (fixed head). Although it may be intuitively clear that the former may be replaced by the latter, the actual reduction is generated by an automorphism of the system onto itself that exploits the queue's circular symmetry. The given map must be checked to ensure it is an automorphism. Of course, reduction represents a one-time cost: the same reduction is used in all subsequent analysis runs for a task until the specification is completely corrected and the task is performed. Therefore, formal specification and verification of reductions was not—and is not expected to be—a major component of testing cost.

Analysis. COSPAN analysis of AUTO Lv1 consists of testing whether the ω-regular language that represents AUTO Lv1 is contained in the ω-regular language that represents the given task (*del* or *seq*). The given task is represented using the same S/R language syntax used to describe the AUTO specification. The *del* and *seq* tasks are defined with one and two lines of S/R code, respectively. However, the *del* task required an additional page of S/R code to describe *exceptions* where performance of *del* is not required (i.e., queue overflow, insufficient timer period, and three other such exceptions).

Once AUTO Lv1 and the *del* and *seq* tasks were defined in S/R and reductions were established, analysis proceeded at the push of a button. COSPAN currently runs on all current versions of several machines, including Digital Equipment Corporation's VAX™, IBM, Amdahl, and SUN computers, and PCs running under various dialects of the UNIX® and MicroSoft MS-DOS® operating systems.

At the start of analysis, a COSPAN run on any of these machines could find errors typically in three minutes of CPU time. Toward the end, when errors were few, a larger computer was required. The larger machines perform analysis at around 10^6 states per CPU-hour in the relevant range. To find the worst error involved a search of over 3 million states and took four CPU hours on the IBM 3080 computer. This search, which was unusually long, was subsequently shortened by applying a more powerful reduction.

Once the final error was eliminated, AUTO Lv1 was proved to perform the *del* and *seq* tasks after searches through under 100,000 states, each search requiring under 3.3 minutes of CPU time. (The search rate worsens as the number of states increase, because of an increased rate of hash conflicts in the table where the reached states are stored.) However, the running time of COSPAN itself was never a significant factor in AUTO's development.

Fault Location. After each run of COSPAN, except for the final ones in which the tasks were proved performed, COSPAN returns the message *task failed* along with some statistics. Most importantly, it returns a sequence of global states of AUTO, from an initial state to a point at which the task failed. The point of failure typically is a cycle of states unaccepted by the automaton that defines the task. Less frequently, the point of failure is a single state where something bad happens, e.g., a state at which a message is transmitted for a second time but with a different sequence number. It should be noted that of all 100 errors in AUTO, not one consisted of a *deadlock*, i.e., a state from where there is no exit. This suggests that, contrary to popular belief, deadlock detection may not be an appropriate protocol-fault model.

Fault location is based on studying the COSPAN-produced tracks from initial states to failures. Often, a single global state in such a track (e.g., a state within an unaccepted cycle at the end of the track) revealed the source of the error. For example, if a pointer has an out-of-bounds value, one may immediately conclude that the pointer is being incorrectly incremented or decremented. But sometimes a small sample of states within an error track reveals nothing obviously wrong. Then, one must "dive into" the track—e.g., around its middle—and proceed through a state-by-state inspection to discover the cause of the fault. In some cases the error track contained too much information to process conveniently. In such cases it is often possible to guess the probable location of the error (e.g., in a certain error recovery module) and then reproduce the error with a simpler track by fixing parameters in areas removed from the error's general location.

In the easiest cases, a fault could be located within a minute of examining the COSPAN error track. In the most difficult cases, it took up to a day or even two to reproduce the error with a manageable track. Such difficult cases, fortunately, were fairly rare. However, more advanced methods of error-track processing would be very desirable. (Some better methods have been developed and used in subsequent projects.)

Development. Most analysis of AUTO was performed at Lv1. After Lv1 was debugged and proved to perform the *del* and *seq* tasks, AUTO was formally refined to Lv2, where implementation of buffering and details of channel interfaces were added. This formal refinement has the property that any task performed by AUTO Lv1 is necessarily performed by AUTO Lv2 as well. Proving that AUTO Lv2 was "legally" refined from AUTO Lv1 required about a week and several minor revisions of AUTO Lv1 and Lv2. (After revising Lv1, the tasks were quickly rechecked there.) AUTO Lv2 was analyzed for the performance of several technical queue-management tasks not relevant to Lv1. This required about another two weeks.

Implementation. AUTO Lv3—the implementation level—was derived from AUTO Lv2 by replacing the three Lv2 processes that model environment interfaces with embedded C-code that performs the input and output with those interfaces. Also, messages abstracted in Lv1 and Lv2 appear as formatted blocks in Lv3. The correctness of the input and output functions performed by the embedded code cannot be verified by COSPAN. However, the embedded code added in Lv3 was less than two pages, about 5 percent of the code forming the AUTO Lv3 implementation. The input/output functions—implemented by reading from and writing to devices—were both simple and standard, and could be expected to work totally or not at all. For this reason, it was not difficult to debug the embedded code. (Initially the read/write commands were faulty, but this was immediately perceived and quickly fixed.)

The major part of the C-code comprising the (Lv3) implementation of AUTO was generated automatically by COSPAN. This implementation, comprising about 1100 lines of code, was 10 percent of the anticipated size, and was considered very efficient. However, some of these automatically generated lines of code were over 250 characters long, underscoring the meaninglessness of this measure of complexity.

Summary. The implementation was basically complete 3 to 6 months ahead of schedule. It was produced essentially by a single S/R programmer and exclusively in the S/R language, except for the small amount of embedded code added at the very end. No errors were found during system test. AUTO Lv1 is somewhat generic, and served as the basis for an auditing protocol at the same interface of TOPAS, thereby substantially reducing the development time of this new protocol. This illustrates our concept of *reuse*, discussed earlier.

Prognosis

A plan is underway to standardize high-level (Lv1) S/R specifications of selected communications protocols. These specifications will be conclusively analyzed, and could be the basis for many, if not all, implementations of these protocols. In addition to the obvious advantage of standardization, this is expected to improve the reliability of the derived implementations while significantly reducing the time required to derive them.

Such an approach may be considered a variation on the quest for reusable software. In this variation, although the implemented software may be very context-dependent and hence unlikely ever to be reused in part or whole, the more generic high-level basis of the implementation may serve in several implementations. This would achieve the same goals normally associated with software reuse, namely increased reliability and decreased development cost.

References

1. J. Katzenelson and R. P. Kurshan, "S/R: A Language For Specifying Protocols and Other Coordinating Processes," *Proceedings of 5th Annual International Conference on Computer Communication*, IEEE, Phoenix, Arizona, 1986, pp. 286-292.
2. R. P. Kurshan, "Reducibility in Analysis of Coordination," *Lecture Notes in Control and Information Sciences 103*, Springer-Verlag, New York, 1987, pp. 19-39.
3. A. P. Sistla, M. Y. Vardi, and P. Wolper, "The Complementation Problem for Büchi Automata, with Applications to Temporal Logic," *Proceedings of 12th International Colloquium on Automata, Languages and Programming, Lecture Notes on Computing Science*, Springer-Verlag, New York, 1985, pp. 464-474.
4. J. R. Büchi, "On a Decision Method in Restricted Second-Order Arithmetic," *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science*, Stanford University Press, Palo Alto, California, 1962, pp. 1-11.
5. R. P. Kurshan, "Complementing Deterministic Büchi Automata in Polynomial Time," *Journal of Computer and System Sciences*, Vol. 35, 1987, pp. 59-71.
6. M. O. Rabin, "Decidability of Second-Order Theories and Automata on Infinite Trees," *Transactions of the American Mathematical Society*, Vol. 141, No. 1, 1969, pp. 1-35.
7. M. O. Rabin, *Automata on Infinite Objects and Church's Problem*, American Mathematical Society, Providence, Rhode Island, 1972.
8. Y. Choueka, "Theories of Automata on ω-Tapes: A Simplified Approach," *Journal of Computer and System Sciences*, Vol. 8, 1974, pp. 117-141.
9. R. P. Kurshan, "Modeling Concurrent Processes," *Proceedings of Symposia on Applied Mathematics*, Vol. 31, American Mathematical Society, Providence, Rhode Island, 1985, pp. 45-57.
10. E. M. Clarke and E. A. Emerson, "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic," *Lecture Notes in Computer Science*, No. 131, Springer-Verlag, New York, 1982, pp. 52-71.
11. Z. Manna and A. Pnueli, "Verification of Concurrent Programs: The Temporal Framework," Stanford University Technical Report CS-81-836, 1981.
12. S. Aggarwal and R. P. Kurshan, "Modeling Elapsed Time in Protocol Specification," *Protocol Specification, Testing and Verification, III*, H. Rudin and C. H. West, eds., Amsterdam, The Netherlands, North-Holland, 1983, pp. 51-62.
13. S. Aggarwal, R. P. Kurshan, and K. K. Sabnani, "A Calculus for Protocol Specification and Validation," *Protocol Specification, Testing and Verification, III*, H. Rudin and C. H. West, eds., Amsterdam, The Netherlands, North-Holland, 1983, pp. 19-34.
14. S. Aggarwal, R. P. Kurshan, and D. Sharma, "A Language for the Specification and Analysis of Protocols," *Protocol Specification, Testing and Verification, III*, H. Rudin and C. H. West, eds., Amsterdam, The Netherlands, North-Holland, 1983, pp. 35-50.
15. S. Aggarwal and R. P. Kurshan, "Automated Implementation from Formal Specification," *Protocol Specification, Testing and Verification, IV*, Y. Yemini, R. Strom, and S. Yemini, eds., Amsterdam, The Netherlands, North-Holland, 1984, pp. 127-136.
16. Z. Manna and P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, Vol. 6, 1984, pp. 68-93.
17. I. Gertner and R. P. Kurshan, "Logical Analysis of Digital Circuits," *Proceedings of the 8th IFIP International Symposium on Hardware Description Languages*, Vol. 8, 1987, pp. 47-67.
18. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, New York, Addison-Wesley, 1979.
19. J. E. Hopcroft, "An $n \log n$ Algorithm for Minimizing the States in a Finite Automaton," *Theory of Machines and Computations*, Z. Kohavi and A. Paz, eds., New York, Academic Press, 1971.
20. P. Halmos, *Lectures on Boolean Algebras*, New York, Springer-Verlag, 1974.
21. A. S. Tanenbaum, *Computer Networks*, Englewood Cliffs, New Jersey, Prentice-Hall, 1981.

(Manuscript received November 17, 1989)