Protocol Validation

The automated, formal validation of software systems continues to be an important area of research. The most significant successes to date with this methodology have been achieved in the area of distributed systems, and the area of data communication protocols in particular. This field was first studied in the late nineteen seventies, initially with rather ad hoc techniques. Today, a well established set of tools and methods has become available to the protocol designer, and can be used to solve routine problems in the design of distributed systems.

Protocol behavior is typically defined as the behavior of an (extended) finite state machine, a definition that can easily be formalized and lends itself readily to the application of automated verification techniques.

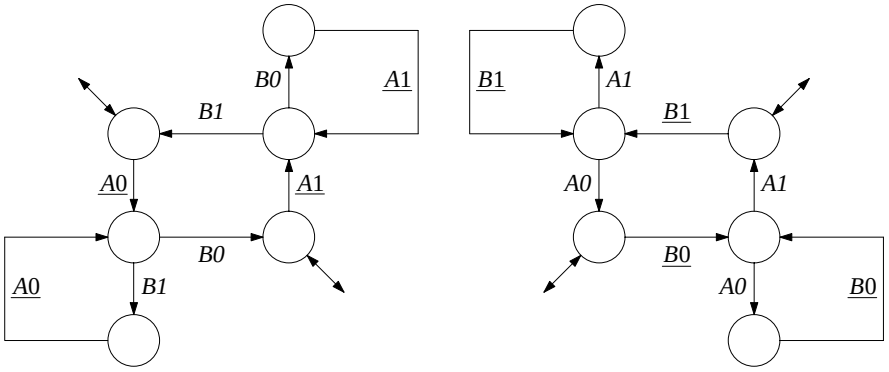

Figure 1 — Alternating Bit Protocol

Figure 1, for instance, show a finite state machine representation of the socalled alternating bit protocol, first defined in this form in 1969 by Bartlett, Scantlebury, and Wilkinson from the National Physical Laboratory in England [1].

Two state machines are defined, formalizing a sender process and an almost identical receiver process. The edge labels in Figure 1 specify message exchanges. Each label consists of two characters, in which the first specifies the origin of the message being received or transmitted, and the second specifies the sequence number that is to be attached to the message. This sequence number was called the *alternation bit*. All underlined names represent send actions; the remaining names represent receive actions. The double arrows, finally, indicate the states where new data can be fetched for transmission by the sender, or accepted and stored by the receiver. or

Since each process can be in no more than 6 process states, the combination of sender and receiver can be in no more than 36 system states. In principle, the exhaustive enumeration of each one of these states can be used to verify that specific correctness requirements cannot be violated in any of these reachable system states.

Background

The first time the automated validation of a communications protocol was tried was by a group at the IBM Research Labs in Zurich [2-4]. They applied a technique, called a ‘perturbation’ or ‘reachability’ analysis, that was first described in 1975 by Carl Sunshine in his PhD thesis [5]. The technique can be seen as a brute force, exhaustive simulation of all possible protocol behaviors, in an effort to prove presence of absence of erroneous or undesirable behaviors. What

should be considered erroneous or undesirable was initially a rather ad hoc set of conditions, such as absence of deadlock or the absence of unspecified receptions (see below).

The subject of the first validation experiments was a simple protocol, CCITT Recommendation X.21. Despite the limitations of the early validation methods, the IBM experiment demonstrated convincingly that even the most rudimentary types of automated validation can almost instantly reveal design errors that expert designers sometimes miss, even after years of study.

Types of Errors Found

Probably the best known type of error that can be found by an automated protocol validation is the possibility of a global system deadlock. A *system deadlock* is a reachable system state from which no other system states are reachable, for instance because all protocol processes are waiting for conditions that can no longer become true. Another well known type of error is the *unspecified reception*. It refers to the possible arrival of a message at a protocol entity, when that entity is in a state where no response for that message was defined. Though these two types of errors are perhaps the best known types of protocol design flaws, they are by no means the only ones, nor the most important ones. One way of classifying errors of this type is to say that they are either caused by the *overspecification* or the *underspecification* of the protocol behavior. An example of the first type is the presence of dead code: code that cannot ever be executed, for instance because the only feasible interactions patterns between the machines prevent it from ever being reached. Examples of underspecification are buffer-overflows, unspecified receptions, and system deadlocks. A more standard, formal classification of errors is based on the terms safety and liveness.

Safety and Liveness

All properties of the types described so far are collectively called *safety properties*. They are necessary requirements on the system behavior, that formalize all the undesirable properties that a protocol behavior should not possess. A mere proof that safety properties cannot be violated, however, is not sufficient to prove also that the protocol will necessarily do what it was designed to do. A separate proof is required to show that something ‘good’ (i.e., the intended purpose of the protocol) inevitably happens. The latter properties are called *liveness properties*. The terms safety and liveness were first defined by Lamport [6].

Requirements for Formal Validation

To perform an automated validation one must formalize two things:

- The protocol behavior proper.
- All correctness requirements on this behavior.

Naturally, both items must be *completely* formalized. There may be no implicit assumptions, specifically not about the way that the protocol interacts with its ‘environment.’ The ‘environment’ of a protocol entity minimally consists of its communication partners and the channels by which it is connected to them.

Protocol Elements

A complete behavior specification includes five basic protocol elements [7]:

1. The *service* to be provided by the protocol to its environment.

2. The *assumptions* that the protocol must make about the behavior of its environment.

3. The *vocabulary* of messages that is used to implement the protocol.

4. The precise *encoding* of each message in the vocabulary.

5. The *procedure rules* for message exchanges.

The first two elements tie the protocol in with its environment. In a layered model (see OSI model) the service (1) defines the interface to the upper protocol layer, and the assumptions (2) define the interface to the lower protocol layer. The last three elements together define a protocol *language* with a vocabulary (3), a syntax (4), and a grammar (5). It is the formal language that is recognized by the protocol entity.

By far most protocol errors are made in the design of the procedure rules. It is amazingly hard to define a logically consistent and complete set of interaction rules for asynchronously executing processes in a distributed system. Alas, human intuition is quite inadequate for correctly locating the potential trouble spots in concurrent systems. Fortunately, automated protocol validation programs have developed their greatest strength in finding the inconsistencies in these types of interaction rules.

Correctness Properties

Safety properties can be formalized as either properties of states, such as *system invariants* or *assertions* about conditions that are required to be met at specific point in a process execution. Liveness properties, on the other hand, are most conveniently formalized as invalid temporal sequences of events. They can be defined, for instance, as restrictions on the set of valid system behaviors. A typical liveness property, for instance, is a progress condition that stipulates that it should be impossible for the protocol to cycle through a sequence of events infinitely often without touching at least one of a predefined set of progress marks in the system (e.g., the increment of a sequence number, or the effective transfer of data from a sender to a receiver). More sophisticated correctness properties can be formalized in a special formal logic that allows one to reason about time sequences. These types of *temporal logic* were first studied as a purely philosophical topic [8]. They were first applied to the analysis of distributed systems by Amir Pnueli in the late seventies [9].

Temporal Logic

Two of the operators that were introduced with the theory of temporal logic are □ (a square box, pronounced ‘always’) and ◊ (a diamond box, pronounced ‘eventually’). The correctness requirement ‘‘always, within a finite time after proposition *p* becomes true, proposition *q* will become true’’ is formalized in temporal logic by the formula □ (*p* => ◊ *q*), where ‘=>’ is standard logical implication.

The validity of a temporal logic formula for a given system behavior can now be checked in two steps. First, the temporal logic formula is translated into a special type of finite state automaton, formally called a Büchi Automaton. This Büchi Automaton is defined in such a way that it can match (i.e., follow, or monitor) only system behaviors that would *violate* the precise correctness requirement that was expressed by the temporal logic formula. That is, the Büchi Automaton formally ‘accepts’ only system behaviors that are violations of the requirement, and the objective of the automated validation is now to prove that the Büchi Automaton will not be able to accept any such behaviors.

Automated verifiers can be used to calculate a minimized Cartesian product of the protocol system with this Büchi Automaton [7,10]. If the product is empty, there are no behaviors in the

protocol system that violate the behaviors that were formalized in the original temporal logic formula (i.e., the correctness requirement). If the product is non-empty, it immediately defines a counter-example to the correctness claim that was expressed.

As an example, consider the temporal logic formula $\Box(p \Rightarrow \Diamond q)$, which expresses the requirement that it is always true that when some proposition p becomes true, eventually some other proposition q will also become true.

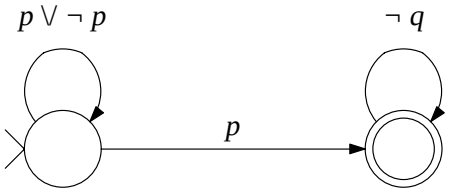

Figure 2 − Büchi automaton corresponding to $\neg(\Box(p \Rightarrow \Diamond q))$

Figure 2 shows the Büchi Automaton that corresponds to the negation of that formula, i.e., the automaton that accepts all behaviors that violate the original formula. The left-hand state is labeled as an initial state for the automaton. The right-hand is marked as an 'accepting' state (with a double circle), expressing the formal requirement that it must be traversed infinitely often by all accepting behaviors. The edges in the graph represent the transitions, and are labeled with the conditions that must be satisfied for the transition to be executable.

By now taking, what is called, the synchronous product of this Büchi automaton with the state machine that formalizes the complete set of behaviors of the system for which the requirement must hold, we can produce either counter- examples to the claim, or, in the absence of any possible counter-example, prove that the correctness claim is satisfied (meaning: it cannot be violated).

Computational Complexity

The main problem that is studied in the area of automated software validation is that of computational complexity, especially the minimal cost that is required to prove the satisfiability of the more sophisticated types of correctness requirements, such as deeply nested temporal logic formulae.

A large class of correctness requirements is formally decidable for protocol systems, provided that they are truly finite state (bounded). Specifically, the formalization of the protocol behavior itself must always produce a bounded system, that can, in principle, be exhaustively searched. Many attractive system properties cannot be adopted in automated validation studies for precisely this reason. If unbounded message channels are used, for instance, all correctness properties of interest (e.g., deadlock or unspecified receptions) become formally undecidable. (A formally undecidable problem provably has no algorithmic solution.)

The standard perturbation analysis exploits the fact that there is always only a finite number of reachable system states. In the worst case all such reachable system states may have to be enumerated exhaustively to prove the satisfiability, or unsatisfiability, of a given correctness requirement. All reachable states together form a graph, usually called the 'system reachability tree,' that completely formalizes all possible system behaviors. Each node in the graph represents a system state, each edge represents the execution of a statement in one of the concurrent processes in the system. The graph can be generated with any of the standard graph exploration algorithms.

The best known algorithm that is used for this purpose is the 'depth-first-search' that was first described by Robert Tarjan [11].

Even though the protocol validation problem for bounded systems, as described above, is decidable, it is by no means efficiently solvable. Even the best possible algorithms have a, socalled, PSPACE (worst case) computational complexity. Very specifically, this means that that amount of computation to be performed to establish the correctness of a protocol may go up exponentially with the number of concurrent processes that together define the system. In the worst case, the *system state space* to be searched is as large as the full Cartesian product of the reachable states of all system components: processes, variables, buffers, and the Büchi Automaton that expressed the correctness requirements. A 32-bit integer variable, for instance, in the worst case has 2^{32} reachable states. Two concurrent counter processes, each counting through the full range of a 32-bit integer, already produce a state space of 2^{32x2} states. Checking each one of these states for a correctness property would be a enormous job. It would take even fastest super-computer centuries of non-stop computation to complete such a chore with a perturbation algorithm. This phenomenon is known as the *state space explosion problem*.

Reduction Methods

Several strategies have been developed to tackle the state explosion problem of a standard perturbation analysis. One method, called *supertrace* [12], is based on an efficient random sampling of very large state spaces, that can proceed several orders of magnitude faster than a perturbation analysis, and provide very high coverage of state spaces that could never be searched with a perturbation algorithm due to the storage requirements. The supertrace algorithms requires just one bit per reachable state to operate.

Several other methods are based on a formal definition of a partial order semantics to remove a large fraction of the provably redundant work that a straight perturbation search incurs [13-15]. In some cases, for the example in the case of the two concurrent counters, these partial order methods can, in effect, replace an exponential complexity with a linear one, and search a state space that is equal in size not to the *Cartesian product* of the two processes, but equal to their *sum*. In more typical applications, the improvement achieved with the usage of partial order rules are a reduction of the state space size by two to eight times. Unfortunately, the problem of computing an optimal reduction based on partial orders can in itself be very expensive computationally (formally, it is another NP-complete problem).

A third class of methods is based on the exploitation of different representations for the system state space, replacing the exhaustive enumeration of all reachable states with the construction of a smaller graph that more efficiently encodes those states. One such encoding is known as a *binary decision diagram*, or BDD, as also used for the encoding of boolean functions in hardware circuitry [16]. As with partial orders, very good reductions can be achieved with BDD's in specific cases. The amount of reduction, however, is usually very hard to predict, and in some cases the 'reduction technique' can even backfire by producing an additional increase of the state space size.

Homomorphism Relations

All three types of reduction methods mentioned above try to avoid the complexity of concurrency during the validation process itself. An altogether different approach is to attempt to reduce the

inherent complexity of the validation model itself, before it is subjected to a formal analysis. Such attempts can be based either on simple 'divide-and-conquer' techniques, or on formal, correctness preserving, simulation relations between abstract and more detailed system components. For the right type of relation, it is possible to replace a complex component for a much simpler one, and thus lower the complexity of the analysis, without altering its validity. Such formal relations are called 'homomorphisms,' e.g. [17].

Like reduction methods, the development of sound theories of effective simulation relations is still an active area of research.

Applications

There have been many successful applications of formal validation techniques to even large industrial design projects. Rudin in [18] describes several such projects performed at AT&T and at IBM. In AT&T's *NewCoRe* project, for instance, over a period of two years (between 1990 and 1992) more than 10,000 formal validation runs were performed by a small team of four 'validation engineers' that was added, on an experimental basis, to a project developing new software for the 5ESS® telephone switch. The team was able to intercept hundreds of subtle errors in the design at an early stage of the design process, clearly demonstrating the viability of modern protocol validation techniques.

Further Reading

An excellent reference to protocols in the context of computer networks is [19]. A comprehensive overview of specific protocol design problems and automated validation techniques can be found in [7]. The SPIN system from [7], for instance, can be obtained free of charge for non-commercial usage, by electronic mail (by sending a message to the internet destination `netlib@research.att.com`), or by anonymous ftp from the machine research.att.com (from directory `/netlib/spin`). More details on temporal logic can be found in [20].

More on the theory of computational complexity can be found in [21]. Specific results for the protocol validation problems were published in, for instance, [22-24]. Two annual conferences publish the latest results on the automated validation of protocols. The first is the IFIP/INWG 6.1 Symposium on Protocol Specification, Testing and Verification (IFIP-PSTV), held yearly since 1981. The proceedings are published by North-Holland Publisher in Amsterdam. The second conference is the Workshop on Computer Aided Verification (CAV), which has been held yearly since 1989. The proceedings are published by Springer Verlag in the series *Lecture Notes on Computer Science*.

Automated protocol validation deals primarily with the logical consistency of the procedure rules of a protocol design. There are of course many other aspects of a communication system that must be analyzed. The most important of these is its real-time performance. The analytical methods that can be used to study such problems are relatively well-known, see for instance [25].

References

[1], Bartlett, K.A., Scantlebury, R.A., and Wilkinson, P.T., ''A note on reliable full-duplex transmission over half-duplex lines,'' *Comm. of the ACM*, Vol. 12, No. 5, 1969, pp. 260-265.

[2], West, C.H., and Zafiropulo, P., ''Automated validation of a communications protocol: the CCITT X.21 recommendation,'' *IBM J. Res. Develop.*, 1978, Vol. 22, No. 1, pp. 60-71.

[3], Zafiropulo, P., ''Protocol validation by duologue-matrix analysis,'' *IEEE Trans. on Communications*, 1978, Vol. COM-26, No. 8, pp. 1187-1194.

[4], West, C.H, ''General technique for communications protocol validation,'' *IBM J. Res. Develop.*, 1978, Vol. 22, No. 3, pp. 393-404.

[5], Sunshine, C.A., *Interprocess Communication Protocols for Computer Networks*, Ph.D. Dissertation 1975, Dept. of Computer Science, Stanford Univ., Stanford, CA.

[6], Lamport, L., ''Proving the correctness of multiprocess programs" *IEEE Trans. on Software Engineering*, 1977, Vol SE-3, No. 2, pp 125-143.

[7], Holzmann, G.J., *Design and Validation of Computer Protocols*, 1991, Prentice Hall, Englewood Cliffs, NJ, 512 pgs, ISBN 0-13-539925-4.

[8], Rescher, N., and Urquhart, A., *Temporal Logic*, 1971, Springer Verlag, Library of Exact Philosophy, ISBN 0-387-80995-3, 273 pgs.

[9], Pnueli, A., ''The temporal logic of programs,'' *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.

[10], Vardi, M.Y., and Wolper, P., ''An automata-theoretic approach to automatic program verification,'' In: *Proc. Symp. on Logic in Computer Science*, pp. 322-331., Cambridge, June 1986.

[11] Tarjan, R.E., ''Depth first search and linear graph algorithms,'' *SIAM J. Computing*, 1:2, pp. 146-160, 1972.

[12], Holzmann, G.J., ''On limits and possibilities of automated protocol analysis,'' *Proc. 7th IFIP WG 6.1 Int. Workshop on Protocol Specification,, Testing, and Verification*, 1987, North-Holland Publ., Amsterdam, pp. 137-161.

[13], Overman, W.T., ''Verification of concurrent systems: functions and timing,'' PhD Thesis, University of California, Los Angeles 1981, 174 pgs.

[14], Holzmann, G.J, Godefroid, P., and Pirottin, D. ''Coverage preserving reduction strategies for reachability analysis,'' *Proc. 12th IFIP WG 6.1 Int. Workshop on Protocol Specification,, Testing, and Verification*, 1992, North-Holland Publ., Amsterdam.

[15], Valmari, A., ''A stubborn attack on state explosion,'' *Proc. 2nd Workshop on Computer-Aided Verification*, 1990, R.P. Kurshan, and E.M. Clarke (Eds.), Rutgers University, Springer Verlag, New York.

[16], Burch et al., ''Symbolic model checking, #10 sup 20# states and beyond,'' *Proc. 5th Symp. on Logic in Computer Science*, Philadelphia, June 1990, pp. 428-439.

[17], Sifakis, J., ''Property preserving homomorphisms of transition systems,'' LNCS 164, 1983, pp. 458-473.

[18], Rudin, H. ''Protocol development success stories,'' *Proc. 12th Int. Symp. on Protocol Specification, Testing and Verification*, June 1992, Fl., North-Holland Publ. 1993.

[19], Tanenbaum, A.S., *Computer Networks*, Prentice Hall, Englewood Cliffs, N.J., 2nd ed., 1988.

[20], Ostroff, J.S., *Temporal logic for real-time systems*, Research Studies Press Ltd, Wiley & Sons, New York, 1989, ISBN 0-86380-086-6, 209 pgs.

[21], Garey, M.G., and Johnson, D.S., *Computers and Intractability: a Guide to the Theory of NP-completeness*, Freeman, San Francisco 1979.

[22], Brand, D., and Zafiropulo, P., ''On communicating finite state machines,'' *Journal of the ACM*, 1983, Vol. 30, No. 2, pp. 323-342.

[23], Apt, K.R., and Kozen, D.Z., ''Limits for automatic verification of finite state concurrent systems,'' *Inf. Processing Letters*, Vol. 22, No. 6, May 1986, pp. 307-309.

[24], Reif, J.H., and Smolka, S.A., ''The complexity of reachability in distributed communicating processes,'' *Acta Informatica*, 1988, Vol. 25, pp. 333-354.

[25], Schwartz, M., *Telecommunication networks: protocols, modeling, and analysis*, Addison-Wesley Pub., 1987, 749 pgs., ISBN 0-201-16423-X.