FORMAL VERIFICATION OF DISTRIBUTED SYSTEMS

INFORMATICA 4/89

Keywords: distributed system, model, formal specification, verification

Tatjana Kapus, Bogomir Horvat
Tehniška fakulteta Maribor

Distributed systems are inherently concurrent, asynchronous, and nondeterministic. Formal methods and automated tools are needed for helping in describing them without causing a misinterpretation, and in reasoning about their correctness. Different approaches to modelling, formal specification and verification of distributed systems are discussed with respect to their abilities. It is difficult to find a universal formal method. Anyway, a formal approach does not have to be universal for being useful in the design of distributed systems.

Za porazdeljene sisteme je značilno hkratno in asinhrono izvajanje komponent ter nedeterministično obnašanje. Zato potrebujemo formalne metode in računalniško podprta orodja, ki bi nam pomagala popolno in nedvoumno opisati sisteme ter sklepati o njihovi pravilnosti. V članku govorimo o različnih pristopih k modeliranju, formalni specifikaciji in verifikaciji porazdeljenih sistemov glede na njihove zmožnosti. Težko je najti univerzalen formalni pristop. Seveda pa je lahko pristop koristen, čeprav ni vsestranski.

0. Introduction

Informal software design techniques often rely on trial and error involving possibly several implementation and redesign loops. Formal methods and computer-aided tools are needed in the entire design process to avoid this potentially expensive procedure. This is especially true for distributed systems, because they are inherently concurrent, asynchronous, and nondeterministic. Conceptually, they are thought of as being composed of processes which interact by exchanging messages. If the number of possible interactions is large, their behaviour is extremely difficult to reason informally about, and even to describe without causing a misinterpretation.

A variety of formal specification and verification approaches are being investigated. They are based on some model of computation. In this paper we discuss different approaches to modelling, formal specification and verification of distributed systems by asking if they have some desirable abilities. We ask, for example, if they manage state explosion, if it is possible to specify and verify safety and liveness properties, if control and data related properties can be verified, if a proof system based on a model is compositional, which kind of communication can be dealt with, if an approach can be used for different applications, and also, if verification can be easily automated.

Note that throughout the paper we talk about the classical system design approach. There, a requirement specification is stated first, which describes the behaviour of a system from its user's view, without talking about its internal structure. It serves as a contract between the user and the designer. A design specification is obtained by decomposing the system into communicating processes. It is the designer's work to verify if it meets the requirement specification. It is good if every process can be specified separately. The possibility of modular specification and verification of a design against a requirement specification in absence of the code reduces the design complexity of distributed systems.

1. State machines

State machines are widely used to model distributed systems. Component processes are represented by states and possible transitions between them. The transitions represent events - transmissions and receptions of messages. The system's state space can be obtained, such that its states are determined by a state of every component, and its transitions are the components' ones. Verification can be generally viewed as requiring reasoning about the complete state space of a system /11/. But, as the number of states increases, it becomes a difficult task. We can say that the basic role of formal techniques is in helping the designer to manage the state explosion.

The problem is being solved in two ways. The first one remains in the state-machine concept. After the component processes are formally specified, the system's state space is constructed and examined. Of course, computers are exploited to do it. This is so-called exhaustive analysis. The state space is in fact a reachability graph, and the analysis is also called reachability analysis. The second way is to use mathematical theories built on appropriate models which would not force us into construction of the state space.

One of the problems with exhaustive analysis is that a representation of the complete state space of a system must be constructed. Usually, some transformations have to be performed to obtain a graph of a reasonable size, such as projections, reductions, and selections /11/. They should preserve the behaviour being analyzed. In most cases, transformations are focused on preserving control aspects, and ignoring data aspects of the system. Unfortunately, only some general properties, such as absence of deadlock or livelock, can be proved in this way, or ordering of communication events can be verified. That is why state machines are typically used for verification of communication protocols. Even when projections are used, construction of a reachability graph is time-consuming for large systems. Besides, finite graphs cannot be built sometimes. One solution to the problem, and to make it possible to analyze more system-specific properties, is the use of simulation as a complementary approach. It is in essence exploration of a selected portion of the state space. However, it cannot "prove" properties about the complete state space, but it can increase confidence in the correctness of the system. A much exploited advantage of simulation is that statistical information about the performance of the system may be calculated from the results of a simulated execution. It is said that "exhaustive analysis and simulation are both sides of the same coin" /2/.

Most currently existent computer-aided tools which can be used for design of real scale distributed systems use exhaustive analysis and simulation. One would say that exhaustive analysis and simulation are used because of the lack of appropriate theories. It is true that for many of them only a conceptual framework is provided, mainly concerning communication and concurrency issues, and their use is only shown for small scale problems. Besides, exhaustive analysis and simulation are certainly more easily automated. In the case of finite number of states, algorithmic verification is possible. But there are other reasons. For instance, it is thought that simulation is nearer to the culture of an average computer scientist /6/. And formal proofs can play its role best when a design is clear enough, while the designer needs tools for assisting him in the design process to achieve this stage by letting him precisely express his ideas, and in the first place validate them rapidly by simulation.

Examples of the tools are OVAL /2/, Véda /6/, RGA /11/, SARA /4/. The tools typically use standardized state-machine languages, such as CCITT's Specification and Description Language (SDL) /2/, and ISO's Estelle /6/, many of them use Petri nets /11/ and related formalisms /4/, for description of systems, because a primary concern here is to provide the designer with a precise and expressive formal language which is easy to learn and to use. This is not the case with abstract mathematical formalisms.

Some analysis procedures, such as searching for deadlocked states, can be built into a tool. A question arises, how to express specific requirements, to traverse the reachability graph interactively, and to verify if they are met. One way is to write them in the same formalism as the design being verified /14/. The RGA tool /11/ allows the user to specify first-order logic propositions and predicates about places and transitions of its Petri-net designs, and even to write an algorithm to perform more complex analysis of the design. Temporal logic specifications may be written in some tools.

There is a similar problem with simulation. In Véda /6/, an observer can be defined to observe execution traces of a simulated system instead of the user, and to report errors when requirement specifications are not met. Another question arises concerning simulation. The development of simulation requires "test scenarios" of the system environment. They can be generated in a fully random or in an interactive way. The authors of SARA /4/, for example, have decided to model the environment explicitly, like the system being designed. A well defined behavioural model of the environment then serves to stimulate the system, and to validate its behaviour. We see that simulation can be in general fully automated.

2. Axiomatic approach

When talking about state-machine notations, we should also mention Milner's CCS (Calculus of Communicating Systems) /9/, and Hoare's CSP (Communicating Sequential Processes) /5/, although they do not model states explicitly. They rather describe processes in terms of observable events. Their advantage is that they provide a range of algebraic laws for comparing, and reasoning about distributed systems, so that formal specification and verification can be carried out in the same framework. If component processes of a system are described in CCS or CSP, we can still construct the system's "state space". This is indeed convenient for "finite-state" cyclic systems, because the observable behaviour of the system can be obtained and simply compared with the system specification for its correctness. But, to avoid a possibly threatening state explosion, other kinds of reasoning have to be employed with state machines, induction on state transitions, for example. CSP offers besides compositional proof rules.

Till now, we have been talking about constructive descriptions of processes. In the constructive approach, also called operational approach, a process is specified as an abstract machine describing a computation. Such a specification is implementation oriented. We specify a program (i.e. a process) essentially by writing another, presumably simpler, program /7/. For requirement specification and formal verification purposes, specifying processes by stating their properties, constraints that any implementation must satisfy, seems more convenient. This is so-called axiomatic approach. In general, there are two kinds of properties. Safety properties express what may happen, or that something bad must not happen. Examples of them are partial correctness, mutual exclusion, and deadlock-freedom. Liveness properties express what must eventually happen, or that a particular good thing must eventually happen. They are temporal properties. Termination and starvation-freedom are liveness properties.

We can talk about the properties in the constructive approach, too. How could we specify a process, if not by describing what may, or what must happen?! Liveness properties cannot be specified in every model. Only safety properties can be stated and verified within the classical state-machine model. If it is not possible to specify what must happen, the second best way is to express what may happen. The difference between the constructive and the axiomatic approach is that in the former we specify a process executions step by step, and in the latter properties are written in form of logic assertions which hold for execution sequences we would get if the process unfolded over time following the constructive description. Unfortunately, algorithmic verification in the axiomatic approach is not possible in general. Proofs have to be designed by hand (if a theorem prover is not available) and a certain ingenuity often is required to find the proof /13/.

Compositionality of proof rules means that proof of the correctness of a compound process can be constructed from proofs of correctnes of its parts. Hence, the system's state space does not need to be constructed. With another word, to prove a property of a program, we do not have to know the complete program, but only requirement specifications of its parts.

Hoare has achieved compositionality by introducing a trace model /5/. A trace of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. A process in the model satisfies a specification if the specification expression is true for all its possible traces. "Concatenation of sequences", "prefix of a sequence", and "the lenght of a sequence" are basic notations to the trace specifications. A similar approach is used in the compositional proof system for networks of processes of Misra and Chandy /10/. It has to be stressed that safety properties hold for complete execution sequences and their finite prefixes. Some liveness properties are not fulfilled by prefixes, but always hold for complete sequences. Because execution sequences may be infinite, liveness properties are difficult to specify in this model due to finiteness of traces.

Temporal logics are most often used for specifying liveness properties. The temporal operator 'eventually' is especially suitable for expressing progress properties, i.e. that an event will eventually happen. Temporal logic is also used in combination with state-machine model in so-called model checking /13/ for verification of liveness properties. Unfortunately, we need here a finite system state space which we check against temporal logic formulae. Finding models that would allow modular verification of temporal properties, and not only of safety properties, i.e. a compositional proof system for both of them, is a tough problem.

An example of such a model is one that has been found by Nguyen et al. /12/. One would expect that infinite sequences will be used in place of finite traces to model process executions. Instead, a behaviour has been introduced for better modelling of progress and termination or deadlock. It is an infinite sequence of observations. Every observation includes a trace of events that have happened up to the moment of the observation. Like in Hoare's traces, the trace may be at most one event longer in the next moment, i.e. in the next observation. It means that events are totally ordered, and that concurrency is modelled with interleaving of concurrent events. The compositional proof system based on the model uses linear temporal logic /8/. With the introduction of infinite behaviours, i.e. infinite sequences of traces, it has been achieved that the previously mentioned trace notations are still the basic ones, but temporal properties can be stated in terms of them by temporal operators. The model is also interesting because systems with synchronous and asynchronous communication can be specified and verified, which is not usual in other existent compositional proof systems. Communication is synchronous if a process cannot send anything until the receiving process is ready to accept it as input, and it is asynchronous if a process can send an output as soon as it is ready. To enable modular specification and verification of temporal properties for both kinds of communication, it has been necessary to represent the readiness of processes to communicate in the model.

It is not necessary to use temporal logic to express temporal properties. It can be always replaced by first-order logic with certain relations introduced. The reason it is often used is because it is concise and elegant.

Trace specifications are very suitable for data-flow computations, for example, but seem awkward in expressing properties whose data structures are not well-defined sequences, such as properties in unreliable systems.

Chen and Yeh /3/ have proposed EBS (Event Based Specification Language) which takes the concept of events more fundamental than that of traces, so that unreliable systems can be more easily specified. Partial ordering on events is used, so that not all possible interleavings of potentially concurrent events have to be considered as with total ordering. And it does not use temporal operators. To say that an event will eventually cause the occurrence of another event, it uses a binary relation. Safety and liveness properties can be specified and verified separately like in Nguyen's system, so that verification is less complex.

3. Conclusion

Some approaches to modelling, formal specification and verification of distributed systems have been discussed. We have shown their main characteristics and problems that have to be overcome in searching for new methods. It seems hardly possible to find a universal formal approach. A much exploited solution is in building automated tools which integrate several useful approaches, and do not force their users into procedures unnatural to them. And perhaps it is true that theories for specific applications should be established before going into generalization /1/.

References

/1/ Boute, R.T. (1988), "On the shortcomings of the axiomatic approach as presently used in computer science", CompEuro 88, System design: Concepts, methods and tools, Brussels 1988, Washington 1988, pp. 184-193.
/2/ Cavalli, A.R. and Paul, E. (1988), "Exhaustive analysis and simulation for distributed systems, both sides of the same coin", Distributed Computing, vol. 2, no. 4, pp. 213-225.
/3/ Chen, B.-S. and Yeh, R.T. (1983), "Formal Specification and Verification of Distributed Systems", IEEE Trans. Software Eng., vol. SE-9, no. 6, pp. 710-722.
/4/ Estrin, G., Fenchel, R.S., Razouk, R.R., and Vernon, M.K. (1986), "SARA (System ARchitects Apprentice): Modelling, Analysis, and Simulation Support for Design of Concurrent Systems", IEEE Trans. Software Eng., vol. SE-12, no. 2, pp. 293-311.
/5/ Hoare, C.A.R. (1985), Communicating Sequential Processes, Prentice-Hall International, London.
/6/ Jard, C., Monin, J.-F., and Groz, R. (1988), "Development of Véda, a Prototyping Tool for Distributed Algorithms", IEEE Trans. Software Eng., vol. 14, no. 3, pp. 339-352.
/7/ Lamport, L. (1983), "What good is temporal logic?", Proc. IFIP 83, ed. R.E.A. Mason, North-Holland, pp. 657-668.
/8/ Manna, Z., Pnueli, A. (1981), Verification of concurrent programs, Part 1: The temporal framework, Tech. Rep. STAN-CS-81-836, Stanford University, June 1981.
/9/ Milner, R. (1980), A Calculus of Communicating Systems, Springer - Verlag, Berlin.
/10/ Misra, J. and Chandy, K.M. (1981), "Proofs of Networks of Processes", IEEE Trans. Software Eng., vol. SE-7, no. 4, pp. 417-426.
/11/ Morgan, E.T. and Razouk, R.R. (1987), "Interactive State-Space Analysis of Concurrent Systems", IEEE Trans. Software Eng., vol. SE-13, no. 10, pp. 1080-1091.
/12/ Nguyen, V., Demèrs, A., Gries, D., Owicki, S. (1986), "A model and temporal proof system for networks of processes", Distributed Computing, vol. 1, no. 1, pp. 7-25.
/13/ Pehrson, B. (1989), "Formal Specification Methods", CompEuro 89, Tutorial Sessions, ed. W. Anacker and R. Beyer, Hamburg 1989.
/14/ Rea, K, and Johnston, R.de B. (1987), "Automated Analysis of Discrete Communication Behaviour", IEEE Trans. Software Eng., vol. SE-13, no. 10, pp. 1115-1126.

This work was supported by Research Communities of Slovenia.