E-Process Design and Assurance Using Model Checking

Model checking, an advanced formal method, has potential in helping businesses verify their e-processes. Using a simple online ticket sales example, the authors demonstrate that model checking can feasibly find flaws and errors that testing and simulation often miss.

Wenli Wang
Emory University

Zoltán Hidvégi
IBM Corporation

Andrew D. Bailey Jr.
University of Illinois at Urbana-Champaign

Andrew B. Whinston
University of Texas at Austin

Trust in e-commerce is difficult to establish and maintain. Almost daily, news headlines cover some incident, causing users to question e-commerce systems' trustworthiness. Strong e-process design and implementation is the first line of defense against errors, fraud, and hacking. Minimizing program faults in business operations is critical for an e-business's survival. Carefully designed and implemented code can handle most expected situations that e-processes encounter, so these e-processes often function well within their defined boundaries. But guaranteeing correct processing under all circumstances is extremely difficult, if not impossible. Hidden flaws and errors, triggered only under unexpected, hard-to-anticipate scenarios, lead to subtle mistakes and even catastrophic failures.

Internet businesses often face tremendous uncertainties due to system complexity, rapid development, interconnectivity, and a lack of familiarity with the new technology-based economy. Designing and implementing highly secure and reliable e-processes is daunting. E-commerce managers, developers, and auditors need new tools to assure users that e-commerce systems are secure and reliable.

Through an online ticket sales example, we demonstrate that model checking has potential in economically checking for certain flaws. Model checking is a powerful formal verification method that determines whether a system model satisfies certain specifications under all circumstances. Model checking can locate subtle but critical flaws that conventional design and assurance methods, such as testing and simulation, often miss.

FORMAL VERIFICATION

Unlike testing and simulation, formal verification attempts to span the entire state space and verify every possible combination of inputs. It can also prove certain system properties with mathematical certainty. Verification begins with formal statements about system properties and determines whether every possible system execution satisfies those statements. Formal methods analyze the logic of system processes rather than execute those processes.

Because formal verification rests on mathematical logic and can theoretically account for every possible program execution, it is more complete and reliable than conventional testing and simulation. This type of thorough examination is necessary for critical system properties. In fact, formal methods have detected hardware and software bugs missed by conventional testing methods.[1,2]

Types of formal verification

Formal verification ranges from a manual proof of mathematical arguments to interactive computer-aided theorem proving, and finally to automated model checking.[1]

Manual proofs are time-consuming, error-prone, and often not economically viable. Computer-aided theorem provers help reduce human errors and prove program specifications,[3] but they still require significant expertise and manual hints, which are rarely documented or consistently applied. In addition, when a program has a bug, a theorem prover provides little help in locating it. A theorem prover produces a formal mathematical proof, but in most cases will not

0018-9162/00/$10.00 © 2000 IEEE

prove a statement false without significant help from the user.

Model checkers, on the other hand, are fully automated tools and can verify that a system model satisfies certain properties. If a property does not hold, model checkers can help identify the input sequence that triggers the failure. Automation makes this particular formal method far more attractive to businesses, especially those striving for shorter time to market without sacrificing correctness.

How model checking works

E-businesses run their applications nonstop. Model checking can verify these systems' properties, spanning the infinite behavior of a nonterminating system to verify both *safety* and *liveness* properties. Safety properties are "never" or "always" claims, such as "the number of tickets sold by the ticket server always equals the number of tickets bought by customers." Liveness properties are eventuality claims, such as "we eventually respond to every customer request." Model checking also handles the nondeterminism caused by uncertain communication delays and unknown parameters when concurrent and parallel processes operate over the Internet.

Temporal logic, a precise mathematical formalism, can express temporal properties such as "a certain property holds for all time," "next time," or "eventually."[4] Temporal logic is powerful enough to express properties associated with an e-business's nonterminating, concurrent operations.

Model checking, which uses such logic, treats individual processes as separate state machines; the *system automaton* is the product of these *process automata*. Model checking operates on the system automaton that has finite states. A model checker converts property-specifying temporal logic formulas to state machines called *property automata*. A property automaton runs in parallel to the product of relevant process automata, and its transitions link to events in the product process automaton. A safety property holds if the integrated *process and property automaton* never goes to a bad state—one in which the property is not satisfied, such as when the number of sold tickets does not equal the number of tickets paid for. A liveness property holds if the automaton always returns to a progress state.

To use model checking, a user need only describe the processes and property specifications using a high-level programming language. The model checker automatically translates these processes and property specifications into automata.

Why testing and simulation fall short

Model checking is more reliable than testing and simulation in catching errors. With the latter, each simulation explores a selected execution path of the program from start to finish. As these patches intersect each other, the simulation checks some states several times but completely ignores other states that are not in the paths. For example, a program run may start with a setup phase to initialize the system. Each simulation test case must execute this setup phase before progressing to the main part of the system. A model checker, on the other hand, considers the setup states only once. Then, at the end of the setup phase, it can discover all execution paths starting at that point without having to re-execute the setup for each path.

Another disadvantage of testing and simulation is that the test engineers must select the test cases carefully, updating them every time the model changes to cover new areas in the state space. In contrast, model checkers can automatically search the entire state space.

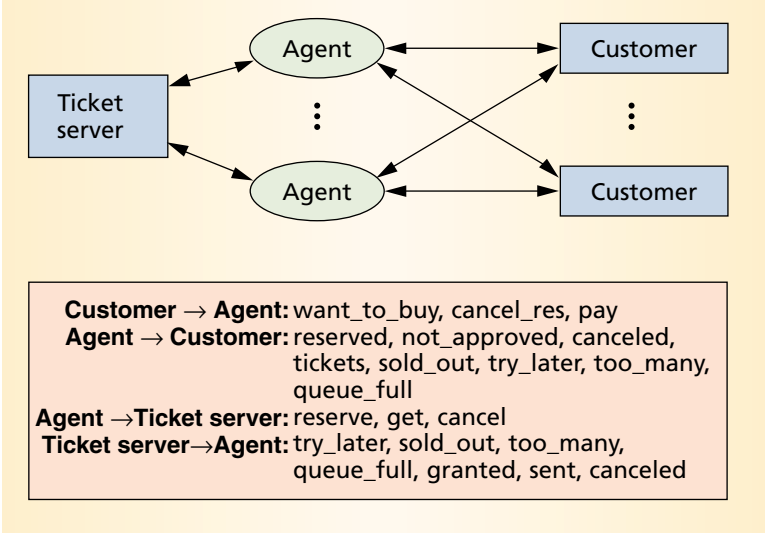

Figure 1. Online ticket sales example.

AN ONLINE TICKET SALES APPLICATION

We demonstrate the feasibility of model checking through an online ticket sales example, a system shown in Figure 1. Although simple, this example embodies all the primary characteristics of an e-business system, including distributed processing, parallelism, concurrency, communication uncertainties, and nonstop operations. Complex e-commerce systems with constrained resources—for example, online stock trading or retailing—exhibit similar features. Insights from our simple example can be helpful in the design and analysis of more complex e-business systems.

All ticketing operations depend on computing processes that execute and interact in a digital Internet environment. The goal is to sell e-tickets with limited quantities over the Internet. A database-implemented ticket server centrally holds all tickets. The ticket server communicates only with intermediary sales agents implemented as automated e-processes written in middleware such as Perl or Java. Customers buy e-tickets through a Web browser from these agents. They can ask the agent to do various operations, and the agent responds by interacting with the ticket server on their behalf. These operations include

When running as a model checker, Spin, like VeriSoft, performs a depth-first search on the state space.

- reserving tickets,
- paying for and getting the reserved tickets, and
- canceling reservations.

The ticket server tracks the number of available, reserved, and sold tickets—denoted as a, r, and s. Let t represent the number of tickets to be reserved in a particular transaction. A sales agent makes a new reservation if $t \leq a$. If $a \leq t \leq (a + r)$, the server responds try_later. The customer can resubmit the reservation later, when the server may accept it if other agents have canceled enough existing reservations. If $0 < (a + r) < t$, the server responds too_many to the agent, and the agent forwards the response to the customer. Once all the tickets have been sold, the agents respond sold_out to any further reservation requests.

The agents are responsible for verifying customer payments. An agent accepts a payment only after notifying the customer of a successful reservation. If an agent rejects a payment, an agent sends the customer a not_approved message and cancels the reservation.

Model checkers applied

We first implemented our ticket sales system in C, using VeriSoft (http://www.bell-labs.com/projects/verisoft/) to verify the system's correctness. Later we switched to Promela, the language of Spin—a model checker that is also well-known as a simulator.

VeriSoft. This model checker can test hundreds of thousands of input combinations in a relatively short time, and can analyze distributed systems written in C and C++. VeriSoft can verify previously written programs, even if the original author did not think about formal verification during coding. Moreover, VeriSoft provides functions that programmers can embed in C/C++ code to increase verification efficiency and coverage.

Programmers can place assertions anywhere in the code using the VS_assert() function. VeriSoft can verify that each assertion always remains true. The VS_toss() function can model external input to the program and nondeterministic events such as message delays. VeriSoft explores a program's possible states using a depth-first search algorithm. The VS_abort() function can prune uninteresting search-tree branches such as execution paths that were previously checked, known to be correct, or uninteresting.

The advantage of using VeriSoft is that it lets us verify an existing C/C++ program with minimal program modifications. VeriSoft also has minimal memory requirements because it doesn't record the states it has visited. Unfortunately, this flexibility does not come without a price. VeriSoft searches the state space only up to a certain depth, which users can specify when running it. Because it does not record the states it has visited, VeriSoft can get into state space loops—revisiting the same state over and over again—making it very inefficient at handling deeper state space searches. In addition, when several paths lead to the same state, VeriSoft evaluates that state for every path. As a result, it may fail to cover the complete state space in a reasonable time even if the state space is relatively small.

VeriSoft provides functions to manually prune execution branches, and this pruning can help eliminate some simple state space loops, but not all. Furthermore, manual pruning makes the verification process less automatic because it requires the VeriSoft user to have a good understanding of the verified system's behavior and state space structure.

Several other formal verification tools avoid these VeriSoft problems by remembering the states already visited. However, recording the states already visited is possible only if the verification program knows the system's exact state space structure. C and C++ are too general to obtain this specification. Therefore, these model checkers usually have their own languages to specify the system model.

Spin. We used Spin (http://netlib.bell-labs.com/netlib/spin/) and its Promela language as an alternative to VeriSoft to verify our model. Because Promela is similar to C, converting our 650 lines of C code into Promela was relatively easy, mainly involving simple changes to conform to Promela's syntax. This conversion took about two hours. The Promela model was richer than the C model yet required only 514 lines of code, providing some indication of the power of Promela.

When running as a model checker, Spin, like VeriSoft, performs a depth-first search on the state space. At each state, it considers all possible transitions but, unlike VeriSoft, executes a transition only if the resulting state has not been visited before. Spin tracks every state it has visited as it executes the system. When it reaches a state it has already visited, Spin backs up to the last decision point (an input or asynchronous event) and tries a different execution path. If Spin has visited all execution paths from a certain point, it backs up until it finds a branch that it has not visited. In this way, Spin can cover all reachable states and considers each state only once.

However, because Spin must remember all the states it has visited, running it requires more memory. Spin has several methods for storing the visited states and has several modes of operation. One mode simply stores each visited state vector in a hash table. The memory required depends on the size of the state vector and the number of reachable states. Spin can require 32 or more bytes per visited state. Another method does not provide 100 percent accuracy but can dramatically reduce the memory requirements. This method creates a large

array of bits or bit pairs, and a hash function maps each state to an element of this array. However, because the array is usually smaller than all possible system states, hash collisions can occur, causing the model checker to skip a branch of execution paths. Nevertheless, this method can still provide high coverage if the number of reachable states is smaller than the size of the array, which should be as big as can possibly fit into the memory. We used a bit array and ran Spin on machines from a laptop with only 80 Mbytes of memory to servers with 2 Gbytes of memory. Spin can run on both, but may miss some problems if there is not enough memory. Spin can also store the state space map in a binary-decision-diagram (BDD)-like structure, which can store the visited states in a very compact format, but this process requires significantly more time.

In addition, Spin can detect nonprogress cycles—infinite execution loops that don't do any meaningful work. Moreover, if the programmer has appropriately placed assertions in the code, Spin can check for assertion violations. Using assertions is good programming practice whether or not you're using formal tools—they're just as useful for simulation. The Promela source code can also contain progress labels, and Spin can verify that all infinite execution loops go through such a progress state.

For instance, in the ticket example, a customer can keep reserving and canceling the reservation indefinitely—an infinite loop—but this is not a bug in our model. A sophisticated system might deny serving such a customer after a certain number of reservation or cancelation requests. To avoid Spin's detecting this condition as a livelock, we marked the cancelation transition with a progress label. Finally, Spin can decide the truthfulness of temporal logic formulas about the model, but since assertions and progress labels adequately described the properties we wanted to verify, we did not use this feature.

Modeling the example system

We used VeriSoft and Spin in the first stage of testing. Then, because of Spin's greater power and efficiency, we used only Spin for subsequent tests.

Also, we had to make several simplifications. First, we considered a system with only two agents and two customers. This simplification preserved the system's interconnectivity, concurrency, distribution, complexity, and nondeterminism. We enforced the following further restrictions to exclude large, uninteresting execution paths from verification:

- *The ticket server has only one ticket for sale.* This might seem like a big restriction, but most problems arise only when the number of available tickets is less than the number of customers, so this situation retains the most interesting execution paths.
- *We do not consider the execution paths after the ticket server's try_later or too_many responses.* This is not a serious restriction: Once these responses reach the customer, the system state becomes the same as the state preceding the original reservation.
- *We consider only one reservation cancellation.* Repeated cancellations do not create new situations.

Our simplified model demonstrates the complexity of e-processes and the feasibility of applying formal verification.

With these restrictions, our system might look extremely simplistic, but VeriSoft still considered 331,079 system states, and the verification (running on a 400-MHz Intel Xeon server) took 28 minutes. This run demonstrates the complexity of e-processes and the feasibility of applying formal verification. VeriSoft helped us identify and fix several programming mistakes very quickly. We found these bugs without writing specific system tests. Most of the bugs were small and trivial, but even these can easily escape testing and simulation if they reside in a branch of code that the tests don't execute. For every bug VeriSoft found, it produced a trace of events leading to the problem, so that we could identify and fix the bug quickly.

Finding the bugs

In the first-stage test, we modeled the customer process as well behaved—one customer transaction at a time. After submitting a reservation, the customer waited for the confirmation or rejection. If the system confirmed a reservation, the customer either paid or canceled. As you might expect, we found no bugs in this predictably stable situation. In the second stage, relying on Spin, we found two bugs when we introduced somewhat "abnormal" or irrational customer behavior—not unusual for Internet users.

Deadlock. We modeled a cracker process that submitted reservations continuously even if the cracker had several reservations pending. Spin identified a design flaw resulting in a deadlock. Its graphic tool, XSpin, helped us pinpoint the scenario causing the deadlock. Figure 2 depicts a system with one agent and one cracker.

Process 0 does no real work but initiates the agent:1, cracker:2, and TicketServer:3 processes. At state 58, the agent sends a message to the cracker, but the cracker does not read it. Just before state 63, the ticket server sends a try_later message to the agent. At state 63, the cracker tries to send a message to the agent, but the communication queue (which, in our model, can hold only one message) blocks it because the agent has not yet read the ticket server's try_later message. The agent is trying to send a message to the cracker, but the communication queue blocks that message too because the

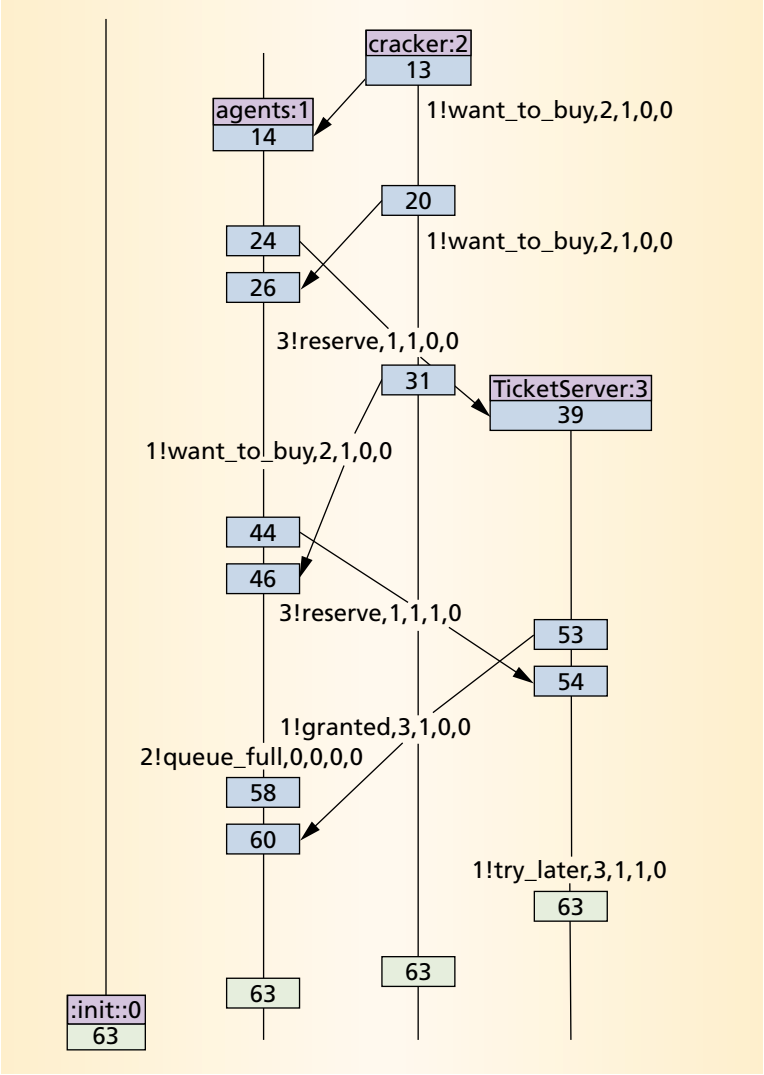

Figure 2. A scenario in the ticket sales example that causes a deadlock (based on a Message Sequence Chart window of XSpin). Vertical lines represent processes, boxes represent states, and arrows represent messages that have been sent by a process and received by the addressee. Each message appears as a "target!content" label. If a message is only sent, and never received, the label appears near the sender. If the message is received, an arrow goes from the sender to the receiver, and the label is near the arrow. The figure denotes messages by the number of the receiving process, an exclamation point, and the message content. The message content includes the request or reply, number of the sending process, and other information.

cracker has not yet read the agent's message sent at state 58. Therefore, the agent and the cracker are waiting for each other, resulting in an infinite wait loop.

Deadlocks are among the most typical problems in distributed systems. They are often difficult to find and reproduce. Crackers can often exploit a deadlock for a denial-of-service attack. Although you might think crackers wouldn't benefit directly from such attacks, they can blackmail the business, conduct industrial espionage, use masquerade techniques more effectively when the legitimate system is down, or trigger problems for interconnected businesses.

After analyzing this deadlock scenario, we introduced time_out to break the loop. Using time_out is quite common in designing networking protocols, but e-business application designers and programmers have not given this technique much attention when designing distributed e-business applications.

Specification flaw. Spin also identified a minor flaw in our original specifications. The problem occurs when the system has only one customer and that customer wants to buy only one ticket but makes two reservations simultaneously with two agents.

Agent A, handling the customer's first request, contacts the ticket server, learns that no more tickets are available, and notifies the customer. Hearing the bad news, the customer leaves that ticket server. During this period, agent B, having been overloaded or blocked, does not notice the customer's second reservation request. When agent B does notice this request, it forwards it to the ticket server, but the request remains unanswered because the ticket server closes when the system has no more customers. This action violates the original property, "every request is eventually responded to." We, therefore, refined the system specifications to accommodate such situations.

Testing and simulation are unlikely to find these types of flaws. E-business designers and developers probably wouldn't consider this scenario, because under normal conditions all agents would have similar processing speed. But in the real world, unexpected communication delays or a DoS attack can generate such a scenario.

The fact that we uncovered these significant flaws using even this simple model shows that the approach we advocate can yield measurable business value for a reasonable amount of effort.

E-processes are the brains and nerves of an e-business. Used correctly and throughout the system development phase, model checking can help locate and correct even subtle but potentially crucial flaws and errors in these e-processes. Such problems are difficult or even impossible to identify using conventional testing and simulation.

Nevertheless, formal verification is not a panacea. Though very useful, it still cannot provide a 100 percent guarantee of system correctness. Limiting factors include the

- business system's complexity,
- difficulty of modeling business systems,
- limited expressiveness of formal presentation language, and
- complexity and cost of current reasoning procedures.

Because of the high cost, we suggest first applying model checking to mission-critical or pervasive business applications where the potential payoff is greatest.

Future research should apply economic reasoning, such as mechanism design, to develop proper online trading rules that are robust against new types of Internet fraud.[5] An integrated approach to e-process design and implementation—combining economic theories and computer advances—is the best way to foster growth in e-commerce in the face of potentially damaging challenges. ✸

Acknowledgments

This research is sponsored in part by grants from IBM Research and Intel. We thank J. Strother Moore, E. Allen Emerson, and Robert P. Kurshan for their help on this project. Both Moore and Emerson are computer science professors at the University of Texas at Austin. Kurshan is a researcher with the Computing Sciences Research Center at Bell Laboratories in Murray Hill, New Jersey. We would also like to thank the reviewers for their comments.

References

1. R.P. Kurshan, *Computer-Aided Verification of Coordinating Processes*, Princeton University Press, Princeton, N.J., 1995.
2. R.P. Kurshan, "Formal Verification in a Commercial Setting," *Proc. Design Automation Conf.*, ACM Press, New York, 1997, pp. 258-262.
3. M. Kaufmann, P. Manolios, and J. Moore, *Using the ACL2 Theorem Prover: A Tutorial Introduction and Case Studies*, Kluwer Academic Publishers, Boston, 2000.
4. E.A. Emerson, "Temporal and Modal Logic," *Handbook of Theoretical Computer Science*, Elsevier Science, Oxford, UK,1990, pp. 997-1071.
5. W. Wang, Z.T. Hidvegi, and A.B. Whinston, "Designing Secure Mechanisms for Online Processes," to appear in *Proc. Int'l Conf. Electronic Commerce 2000, Seung Leem Publishing, Chung-gu, Seoul, Korea, 2000, pp. 312-318.*

Wenli Wang *is a visiting assistant professor of decision and information analysis in the Goizueta Business School at Emory University, Atlanta. Her research interests focus on electronic commerce security, control, and assurance. Wang has a PhD in management information systems from the University of Texas at Austin. Contact her at Wenli_Wang@bus.emory.edu.*

Zoltán Hidvégi *is a staff software engineer at IBM Corporation in Austin, where he works on functional verification tools. He is also pursuing an interdisciplinary PhD between computer science and management information systems at the University of Texas at Austin. His research interests focus on formal verification. Hidvégi has an MS in mathematics from Eötvös University in Budapest, Hungary. Contact him at hzoli@austin.ibm.com.*

Andrew D. Bailey Jr. *is the Ernst & Young distinguished professor of accounting at the University of Illinois at Urbana-Champaign. His research interests include statistics and auditing in a computerized environment. Bailey received a PhD in accounting from Ohio State University. Contact him at jabaile@uiuc.edu.*

Andrew B. Whinston *is a professor of information systems, economics, and computer science; the Hugh Roy Cullen Centennial Chair in business administration; and the director of the Center for Research in Electronic Commerce at the University of Texas at Austin. His research interests span a range of issues in electronic commerce, including resource allocation, bundle markets, trust, assurance, and market design. Whinston has a PhD in management from Carnegie Mellon University. Contact him at abw@uts.cc.utexas.edu.*