DESIGNING RELIABLE DISTRIBUTED SYSTEMS

A formal language for modeling distributed systems

STEPHEN J. GARLAND

Computer systems fail, often with costly results. Even simple errors (division by zero and buffer overruns, for instance) that can be detected easily by code inspection or testing cause havoc.

More pernicious and harder to detect are timing bugs, which can lie dormant until triggered by some unusual combination of circumstances. In the mid 1980s, for instance, several cancer patients died from massive radiation overdoses from the Therac-25, a computerized therapy machine. At fault was improper synchronization between data entry and dose calculation, which failed when operators typed more quickly than had the testers. In 1990, 60 million AT&T subscribers lost telephone service for nine hours (and AT&T lost $60 million in revenue) after an electronic switch failed and set off a chain reaction in which other switches crashed, rebooted, and crashed again. More recently, software guided the Mars Pathfinder to a spectacular landing, but experienced total system resets when it began to collect—but could not transmit—meteorological data. At fault in this case was a latency problem in a preemptive priority-scheduling routine.

Computer systems such as these are providing increasingly powerful services, and society is expecting increasingly strong guarantees of performance, fault tolerance, and security. At the same time, however, the environments and networks in which these systems operate are becoming larger and less predictable. It is no wonder that distributed systems, which consist of many programs running on many different computers connected via a network, have become very complicated and very difficult to get right. For years, testing has been the primary means of ascertaining whether programs behave as intended. However, as Edsger Dijkstra

***Stephen is a Principal Research Scientist in the MIT Laboratory for Computer Science. He can be contacted at* garland@lcs.mit.edu.**

wrote in 1972, "testing can be used to show the presence of bugs, but never to show their absence." So, as programs become more complicated, it is high time to augment testing with other techniques for designing and reasoning about complicated systems.

Unfortunately, it is very hard to reason about most distributed systems in full detail, as is apparent to anyone who has tried to reason carefully about a large software system. One reason this task is so difficult is the paucity of modeling techniques and tools: Computer systems are unique among engineering artifacts in that their development rarely employs predictive mathematical models. Another reason is that, even with adequate models, proofs about programs can be tedious. Doing them by hand is cumbersome, boring, and prone to mistakes. Unless proofs are machine-generated or machine-checked, there is little reason to believe them. And unless the models used for proofs are tied to the languages used for programs, there is little reason to believe that the proofs have anything to do with the programs.

For several years, computer scientists at MIT have been attempting to address these problems by designing models, languages, and tools for constructing reliable distributed systems. Related efforts are underway at universities, research labs, and companies such as Cornell, the University of Texas at Austin, SRI, AT&T, Motorola, and Intel. All these groups aim to make software engineering more like other engineering disciplines through a use of mathematical models and methods.

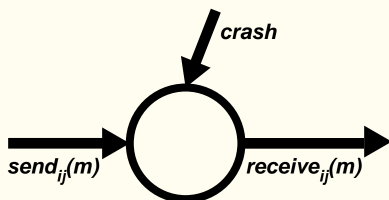

Figure 1: *Communication channel.*

The current focus of our work at MIT is on IOA, a simple formal language for modeling distributed systems (http://theory.lcs.mit.edu/tds/ioa.html). IOA, which is written in Java, lets system designers express their designs at different levels of abstraction, starting with a high-level specification of required global behavior and ending with a low-level version that can be translated easily into real code. An accompanying IOA toolset provides access to a range of validation methods, including machine-checked proofs and simulation. IOA tools let designers reason about properties of their designs at all levels of abstraction and establish relationships between different levels. An additional code-generation tool will soon connect verified low-level designs to executable distributed code, thereby carrying over claims and proofs about designs to real distributed programs in programming languages like C++ and Java. IOA's combination of verification and code-generation tools is intended to ensure that the final programs are correct, subject to explicitly stated assumptions about the behavior of externally provided system components (for example, about the behavior of standard communication services).

Modeling Distributed System Components

The theoretical community has long modeled distributed systems as collections of reactive components; that is, of interacting state machines. Input/output (I/O) automata, developed by Nancy Lynch and Mark Tuttle at MIT, and statecharts, developed by David Harel at the Weizmann Institute, are two such formalisms. Each can be used to model system components that interact with each other and that operate at different speeds. Each has mechanisms for composing modeled components into larger systems. Statecharts have an appealing graphical representation, but I/O automata are more amenable to formal proofs.

An I/O automaton is a simple type of state machine in which transitions from state to state are associated with named actions. The actions are classified as input, output, or internal. The inputs and outputs are used for communication with other automata, whereas internal actions are visible only to the automaton itself. The automaton controls which output and internal actions are performed, but not which input actions occur; other automata initiate these actions, and the automaton cannot block them from occurring.

For example, Figure 1 shows a communication channel between two nodes, *i* and *j*, in a distributed system. Node *i* can place a message *m* in the channel via a *send* action, after which the channel can deliver the message via a *receive* action to node *j*. An additional *crash* action, generated by the environment, can affect the behavior of the channel. The circle represents the channel, an incoming arrow an input action, and outgoing arrow an output action. Figure 1 does not show the channel's internal actions, which may divide messages into packets, forward them through routers, and reassemble them for delivery to *j*.

Central to the I/O automaton model is its trace-based notion of external behavior. Traces are permissible sequences of input and output actions, which serve to distinguish one kind of automaton or communication channel from another. A reliable first-in/first-out (FIFO) channel, such as that provided by the standard Internet TCP protocol, neither loses nor reorders messages. An unreliable channel, such as that provided by the UDP protocol, may lose or reorder messages. Thus,

the traces *…send1,2 (A)…send1,2 (B)… crash…receive1,2 (B)…send1,2 (A) …send1,2 (B)…receive1,2 (B)…send1,2 (C)…receive1,2 (A)…* can be traces of UDP, but not of TCP. Other kinds of channels may duplicate messages or even inject spurious ones. Defining external behavior in terms of traces lets us view systems at varying levels of abstraction, compare them by comparing their traces, and describe the behavior of a composite system by identifying the inputs of some components with the outputs of others, and then by interleaving the traces of the components.

Automata produce r external behavior by performing transitions that affect their internal states. Figure 2 shows the specification of a reliable FIFO channel in IOA. The specification says nothing about how the channel is implemented to ensure reliable delivery. It simply requires that messages be received in the order they are sent, with no duplicates or omissions.

```
automaton channel(Node, Msg: type, i, j: Node)
  signature
    input send(m: Msg, const i, const j)
    output receive(m: Msg, const i, const j)
    input crash
  states buffer: Seq[Msg] := {}
  transitions
    input send(m, i, j)
      eff buffer := buffer |- m
    output receive(m, i, j)
      pre buffer  {}  m = head(buffer)
      eff buffer := tail(buffer)
    input crash
```

Figure 2: *IOA specification of a reliable communication channel.*

The channel automaton in Figure 2 is parameterized by two data types, *Node* and *Msg*, which can be instantiated to describe the identifiers for communicating nodes (that is, IP addresses or domain names) and the messages that can be sent (strings or numeric data). It is also parameterized by two identifiers, *i* and *j*, for the sending and receiving nodes. Its signature contains a single *send/receive* action for each *m* in *Msg* and a single *crash* action. The keyword *const* indicates that the values of *i* and *j* in these actions are fixed by the values of the automaton's parameters. Its state consists of a single variable, *buffer*, which initially contains an empty sequence of messages.

The transitions of channel are described in terms of their preconditions and effects. A precondition is a predicate on the state indicating the conditions under which an action can occur (for example, *buffer* must be nonempty and have message *m* at its head for a *receive* action to occur). Input actions have no preconditions because they can occur at any time. The effect describes the changes that occur as a result of the action, either in the form of a simple program (appending *m* to *buffer* or removing *m* from its head, for example) or in the form of a predicate relating the states before and after the action occurs. At the level of abstraction in Figure 2, the *crash* action has no effect on the state of a reliable channel.

Specifications for less reliable channels differ from that in Figure 2 in any number of ways. Figure 3 illustrates some of the possibilities. The *crash* action for an in-order at-most-once message-delivery service might simply delete some number of elements from *buffer*, but not change the order of the remaining elements. And the state of an unreliable channel might be represented as a set rather than a sequence, and the *crash* action might delete some number of elements from that set.

Reasoning About Distributed Systems Components

Communication channels play a central role in distributed systems. It is important that client programs understand—and rely on no more than—the guarantees they provide. And it is important that a channel's implementation actually provides those guarantees. System designers often balance the behavior a communication channel or other component guarantee against its performance costs. Strong guarantees, such as those provided by a reliable FIFO channel, may be essential for electronic funds transfer. However, weaker guarantees may be more appropriate for applications where latency (the time it takes to deliver a message) is more important than absolute accuracy. For example, people using Internet telephony may not be bothered if static obscures occasional syllables, but may become impatient if the network interjects long pauses in their conversation. For applications like this, it makes sense to sacrifice accuracy for speed.

Before the development of IOA, people at MIT were designing an efficient message-delivery service that was allowed to lose occasional messages. They proposed a simple abstract specification of the service that would be easy for clients to understand—the one just cited in which a crash can delete some number of elements from *buffer*, but not change the order of the remaining elements. In IOA, a simple definition

input crash
eff buffer := choose b where b buffer

for the *crash* action was enough to describe such a "lossy" channel. To use

it, you only had to provide axioms for a subsequence relation () to supplement IOA's built-in definitions for the operations head, tail, and append (|-). The designer's distributed implementation of the lossy channel was considerably more complicated than this specification—the sending node would periodically retransmit sequence-numbered messages until the receiving node acknowledged receipt or receipt of a higher numbered message. But the designers saw no reason to confuse clients with these details.

The designers were not particularly happy, however, with a long and complicated proof constructed to show that their implementation met their specification. They feared that even small programming changes, designed to increase efficiency, might break the proof if not the entire implementation. Hence, they sought advice from an expert in verification to find a simpler proof—one less sensitive to changes in the implementation. The expert's response was disappointing: Their proof was complicated, he claimed, because they were using the "wrong" specification! According to the expert, messages in transit when a crash occurs might not be lost, only delayed; hence, a more honest and tractable specification would represent the loss of a message as a separate action, which could occur subsequent to a crash.

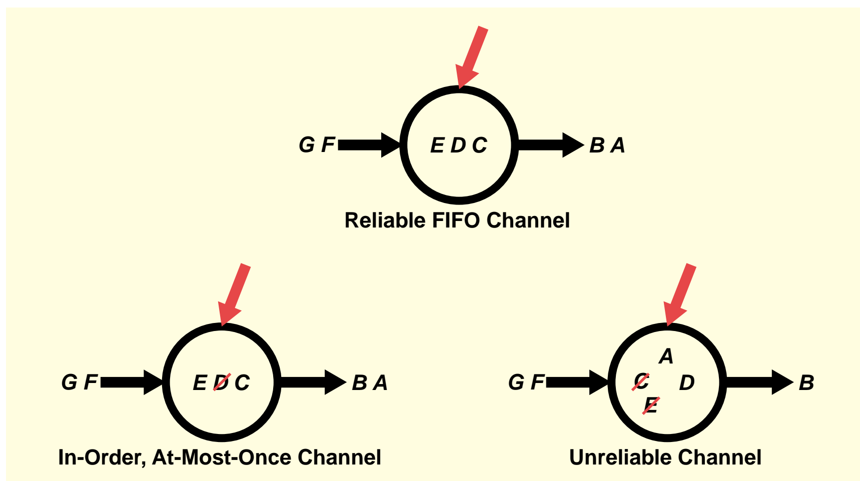

Figure 3: *Different communication channel behaviors.*

```
axioms MarkedMessage for Mark[__]
automaton honestSpec(Node, Msg: type, i, j: Node)
  signature
    input send(m: Msg, const i, const j)
    output receive(m: Msg, const i, const j)
    input crash
    internal lose
  states buffer: Seq[Mark[Msg]] := {}
  transitions
    input send(m, i, j)
      eff buffer := buffer |- [m, unmarked]
    output receive(m, i, j)
      pre buffer  {}  m = head(buffer).msg
      eff buffer := tail(buffer)
    input crash
      eff buffer := markAllMessages(buffer)
    internal lose
      eff buffer := deleteSomeMarkedMesages(buffer)
```

Figure 4: *Revised IOA specification of a lossy communication channel.*

Figure 4 is the considerably more complicated "honest" specification formulated in IOA. In that specification, the *crash* action does not result in the immediate loss of messages from the queue; rather, it marks messages as losable by subsequent internal actions. In addition, new axioms defined new operators such as *markAllMessages* and *deleteSomeMarkedMessages*. This was hardly a specification the designers wanted to show to users.

Fortunately, the designers were able to have their cake and to eat it too: The two specifications were actually equivalent in that they had the same set of traces. Once this equivalence had been established, the designers could show the simpler specification to clients, and use the more detailed specification to simplify their original proof that their implementation was correct.

How was equivalence established? Two proofs were involved, one to show that every trace of the desired user specification (specification *A* in Figure 5) was also a trace of the honest specification (specification *B*), and another to show trace inclusion in the other direction. The first proof involved the construction of a so-called forward simulation relation between the states of *A* and those of *B*. As shown in Figure 5, such a relation *R* must satisfy two conditions:

- Each initial state of *A* (shown in black) must be *R*-related to some initial state of *B* (shown in red).
- For each transition (*sA*, *a*, *sA*) of *A* and each state *sB* of *B* such that (*sA*, *sB*) is in *R* there must be an execution fragment (that is, a

sequence of transitions) of *B* that corresponds to the given transition in a particular way; namely, it has the same trace and leads to a state *sB'* with (*sA'*, *sB*) in *R*. Given such a simulation relation *R*, it is easy to map, step by step, any trace of *A* to a trace of *B*.

For the lossy channel, the required simulation relation simply throws away the marks accompanying each message in *B*'s buffer. This relation is easy to define in IOA: *forward simulation from A to B: A.buffer = messages(B.buffer).*

The proof that this relation has the required properties is also easy: The *send* and *receive* actions in *A* and *B* correspond in the natural way, and *A*'s *crash* action corresponds to a crash followed by an immediate loss in *B*. In fact, the proof is so easy that it takes little effort to get LP (an automated proof assistant developed at MIT; see http://nms.lcs.mit.edu/Larch/LP/overview.html) to check every detail; LP required but a single hint from users to find the proof.

The second proof, that every trace of *B* is also a trace of *A*, requires a more elaborate notion of a simulation relation, a more complicated definition of the simulation relation: *backward simulation from B to A: s (A.buffer = messages(s) subseqMarked(s, B.buffer)),* is a longer proof that the definition has the required properties, and more hints (about a dozen) to guide LP through the proof. But once done, the two machine-checked proofs made it clear that the two specifications for the lossy channel were really equivalent.

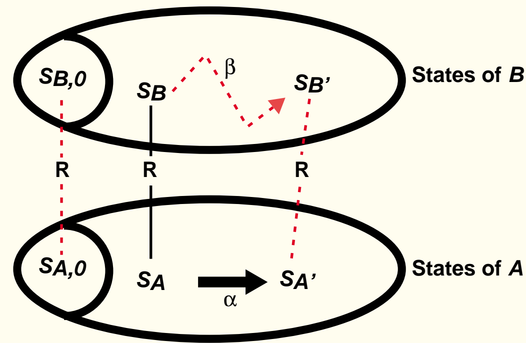

Figure 5: *Requirements for a forward simulation relation.*

The IOA Language and Toolset

This design experience motivated the development of IOA at MIT. It showed that I/O automata could be used to help in the design of real systems, and also that proofs based on I/O automata were well suited to computer-assisted verification. At the same time, it showed the need for a new language, because practical use of I/O automata demanded tool support. Tool support, in turn, demanded a language for describing I/O automata.

Designing the IOA language was a challenge because it had to support both proofs and compilation of executable code. These twin demands created a tension in the design because features that make languages suitable for proofs—a declarative style, simplicity, and nondeterminism—differ from those that make them suitable for code generation—an imperative style, expressive power, and determinism. Nondeterminism is good for proofs because it lets designers validate general designs; for example, designers can describe an at-most-once message-delivery service without having to specify which messages, if any, are dropped. Furthermore, a simple language with a declarative style is easiest to translate into the input languages of standard theorem provers and to manipulate in interactive proofs. On the other hand, programmers prefer more expressive languages, and a deterministic language with an imperative style is easiest to translate into executable code.

To meet its twin objectives, IOA permits transition definitions that use simple programs, predicates, or a combination of the two. Furthermore, accompanying tools support transforming specifications from one form into another, depending on whether the task at hand is verification or code generation. Since the I/O automaton model is a reactive system model rather than a sequential program model, the IOA language reflects this fundamental distinction. That is, IOA is not a standard sequential programming language with some additional constructs for concurrency and interaction; rather, these concepts are at its core.

The IOA language is accompanied by a set of software tools, which are designed to make IOA easy to use. Basic tools include a prettyprinter for producing easy-to-read listings, and a static checker for detecting syntactic and semantic errors that might indicate deeper problems with a specification or might confuse a reader.

A simulator runs sample executions of an IOA program, letting users help select the executions. It helps users perform sanity checks, by letting them test purported properties of I/O automata before attempting to prove them. A novel kind of paired simulation aids checking purported simulation relations. (Somewhat confusingly, IOA inherited two different senses of the word "simulation" from other areas in

computer science.) A paired simulation checks whether a candidate relation R can be a simulation relation from A to B by simulating A as usual, and using its execution to generate a corresponding execution of B. The simulation also checks that B's actions satisfy their preconditions, that explicit nondeterministic choices satisfy required constraints, that the simulating fragment has the same external behavior as the given transition, and that the relation R holds between the states of the two automata after the transition.

Interfaces to existing proof tools support:

- Proving lemmas and theorems about properties of data types used to define automata.
- Checking the conditions of formation for I/O automata (for example, that the sets of input, output, and internal actions are disjoint).
- Proving invariants of automata (properties true in every reachable state).
- Proving simulation relations between automata.

Some of these properties can be checked statically for some automata, but many require theorem-proving assistance. Proof tool interfaces can suggest lemmas and proof tactics that existing proof tools would not discover on their own.

Experience shows that the benefits of machine-checked proofs outweigh the additional effort required to construct them. The principal advantage is that machine-checked proofs are more reliable. They are harder than manual proofs primarily where manual proofs are vague and, therefore, suspect. An additional benefit is that careful checking helps you see exactly what is necessary to ensure the desired properties. At times, this leads to simplifications in a system's design. A final benefit is that machine-checked proofs are easy to repeat, particularly for regression proving after a small change in design requirements or implementation. Without machine assistance, programmers are likely to skimp on such proofs, believing that the change couldn't possibly affect properties that are tedious to recheck.

Finally, a still-to-come code generator will translate low-level IOA specifications (produced and verified by the user with the aid of other IOA tools) into Java code that runs on a collection of workstations communicating via standard networking protocols such as MPI. The code generator will help overcome the current problematic and widespread disconnect between formal specifications and system implementations. Central to its operation are multiple models for external system services. Abstract models describe interfaces and behaviors of communication channels that system designers want to use (for example, reliable FIFO channels). Lower-level concrete models describe the actual interfaces and behaviors of existing external services (TCP or MPI, for instance). The code generator provides auxiliary IOA automata for each external service that, as in Figure 6, combine either with the service's concrete model to implement a desired abstract channel or with an IOA program designed to run on nodes of the distributed system. In so doing, it generates code that is guaranteed to use the external service correctly.

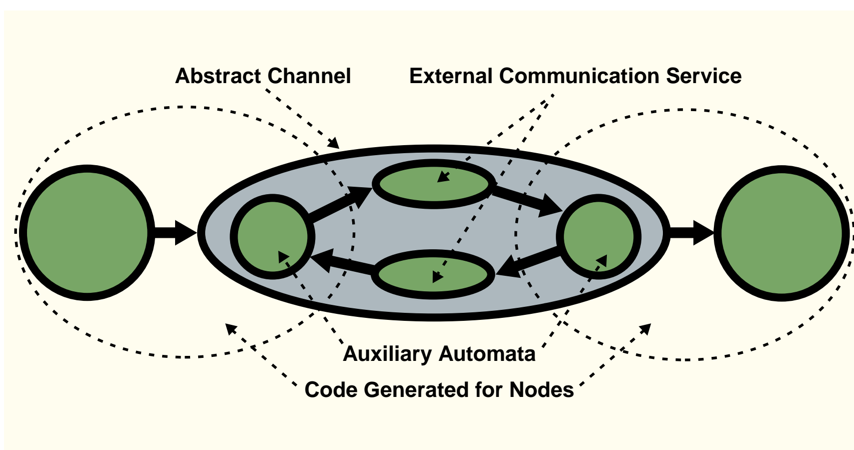

Figure 6: *Code generation strategy.*

Building Systems Out of Components

In addition to supporting precise descriptions for system components, such as communication channels, IOA also supports precise descriptions for larger systems, obtained by assembling those components. As for components, it also supports reasoning about larger systems at different levels of abstraction.

Consider, for example, a toy banking system in which a single bank account is accessed from several locations, using deposit and withdrawal operations and balance queries. Figure 7 shows an abstract view of the system, in which a central bank interacts with several automatic teller machines, responding to *deposit* and *withdraw* actions by an *ok* action, and responding to a *query* action with a *balance* action after computing the reported balance by an internal *compute* action.

The IOA specification for the ATMs (Figure 8) describes the operations they can invoke and when. The specification expresses a simple condition for the way in which each ATM interacts with the bank, namely, an operation must

complete before the ATM can submit another.

A corresponding specification (not shown) describes what the bank is allowed to do, without any details of the distributed implementation. In response to a *balance* action, the bank automaton performs an internal action to compute the result of some set of prior deposits and withdrawals that includes all operations submitted at the ATM issuing the query; then it reports that result. The response need not reflect *deposit* and *withdrawal* operations submitted at other ATMs. A parameter to the bank automaton indicates the number of ATMs it supports.

The external signature of the bank automaton and the ATM automata are mirror images. *Input* actions of one correspond to *output* actions of the other. This enables the banking system in Figure 7 to be described as a composition of the bank automaton and the requisite number of ATM automata:

automaton AbstractBank(nLocs: Int)
components B: Bank(nLocs);
A[loc: Int]: ATM(loc) where loc < nLocs

IOA's composition mechanism enables the bank and ATM automata to communicate. It identifies actions with the same name and arguments in different components, for example, the input action *deposit(loc, amt)* for *Bank(nLocs)* with the output action *deposit(loc, amt)* for *ATM(loc)*. When any component automaton performs a transition involving an action, so do all component automata that have the action in their signatures.

Composition also facilitates the description of a distributed implementation of the banking system. In that composition, reliable FIFO communication channels, as specified in Figure 2, connect Branch automata, which work locally to process deposits and withdrawals, but send explicit messages to all other branches when they receive balance queries. A Branch automaton collects responses to these messages and combines them with its own known operations to calculate the response to the balance query.

automaton DistributedBank(nLocs: Int)
components B[loc: Int]: Branch(loc)
where loc < nLocs;
L[loc: Int]: ATM(loc)
where loc < nLocs;
C[i, j: Int]: channel(Int, M, i, j) where i < nLocs j < nLocs i ~= j
hidden send(m, i, j), receive(m, i, j)

The *hidden* clause hides output actions of component automata by reclassifying them as internal actions. This prevents them from being used for further communication and means that they are no longer included in traces.

Just as the two specifications for the lossy queue were shown to have the same traces, so too can the *DistributedBank* (Figure 9) be shown to implement the *AbstractBank*; that is, to show that all its traces are also traces of *AbstractBank*. Indeed, a proof of this fact has been checked using LP. However, the proof is quite lengthy, and work is now under way to shorten it

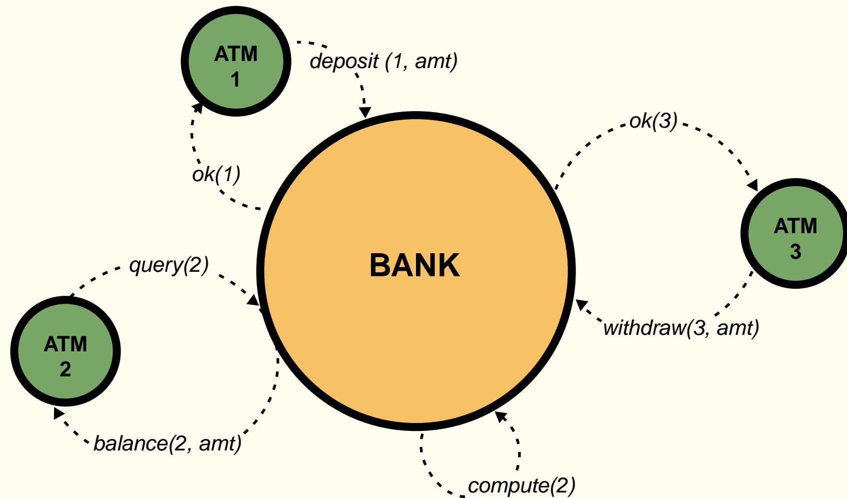

Figure 7: *Abstract banking system.*

```
automaton ATM(loc: Int)
  signature
    input  ok(const loc),
           balance(const loc, amt: Int)
    output deposit(const loc, amt: Int) where amt > 0
           withdraw(const loc, amt: Int) where amt > 0,
           query(const loc)
  states active: Bool := false
  transitions input ok(loc)
                 eff active := false
               output deposit(loc, amt)
                 pre ~active
                 eff active := true
        % ... transitions for other actions defined similarl
```

Figure 8: *IOA specification for ATM machine.*

(and similar proofs) by using simulation to suggest potentially useful lemmas.

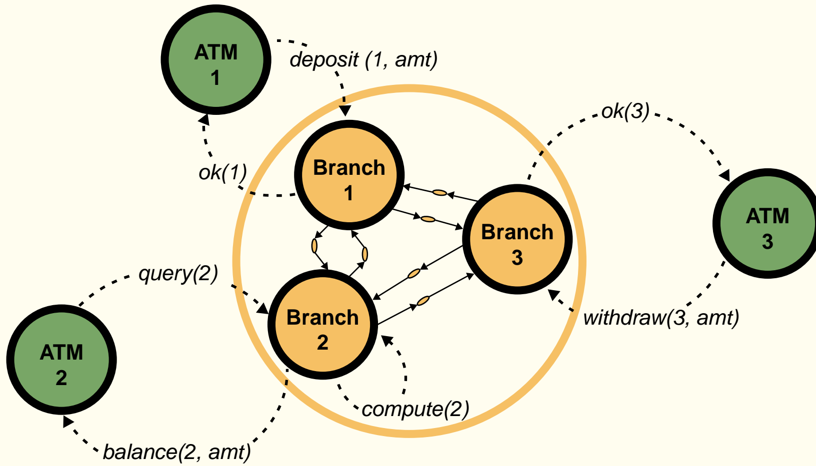

Figure 9: *Distributed implementation of banking system.*

Conclusion

What makes IOA exciting is its potential for assisting in the development of well-engineered software systems not just for banking, but for many other applications. Designers can propose and reason about the effects of using different components and communication mechanisms to build their systems. And when they are satisfied they have the right design, they can generate actual code directly from it without having to fear that programmers will misunderstand the design or introduce transcription errors when coding it.

Languages like IOA and their supporting tools will not make testing obsolete; indeed, simulation in IOA is a form of testing. However, such languages and tools will lead to much more reliable systems than can be achieved by testing alone. ■