Under consideration for publication in Formal Aspects of Computing

A Practical Comparison of Alloy and Spin

Pamela Zave

AT&T Laboratories—Research
Florham Park, New Jersey USA

Abstract. Because potential users have to choose a formal method before they can start using one, research on assessing the applicability of specific formal methods might be as effective in encouraging their use as work on the methods themselves. This comparison of Alloy and Spin is based on a demanding project that exploited the full capabilities of both languages and tools. The study exposed issues not apparent from more superficial studies, and resulted in some unexpected conclusions. The paper provides tentative recommendations for two different classes of network protocol, a research agenda for solidifying the recommendations, and a few general lessons learned about research on selection of formal methods.

Keywords: lightweight modeling and analysis, protocol verification, distributed hash table, Chord

1. Introduction

The field of software engineering has produced an enormous body of work known collectively as "formal methods" for software development. This work builds on the insights of generations of mathematicians and computer scientists, and, to those who are familiar with it, has proved its worth hundreds of times over.

It is a perennial source of frustration that formal methods are not used as often as they should be by software practitioners in industry, and not even by our research colleagues in other disciplines of computer science. Here are the recent comments of a networking researcher, explaining his opinion that analytical approaches have had little practical success [1]:

". . . it is not clear what mathematical approach is the best fit to a given problem. There are a plethora of approaches—each of which can take years to master—and it is nearly impossible to decide, a priori, which one best matches the problem at hand. . . . It all depends on the nuances of the problem, the quality of available tools, and prior experience in using these approaches. That's pretty daunting for a seasoned researcher, let alone a graduate student."

This quotation suggests that research on assessing and explaining the applicability of various formal methods might be as effective in encouraging their use as work on the methods themselves.

A recent survey on the use of formal methods [2] found no correlation between application domains

Correspondence and offprint requests to: Pamela Zave, e-mail: pamela@research.att.com

(*e.g.,* transportation, financial, health care) and techniques used (*e.g.,* specification/modeling, proofs, model checking). The survey found only mild correlation between the type of software (*e.g.,* consumer electronics, transaction processing) and the techniques used. This tells us that knowing current "best practices" will not help a potential user choose a specific formal method to apply to a specific problem.

Although this paper sheds a little light on the selection of formal methods, its primary goal is to encourage and shape further research on this topic, and above all to inspire a sense of urgency. As the paper will reveal, I discovered that an extremely well-known protocol—one that had allegedly been proven correct—has many flaws that could have been found easily with formal modeling. These results have created a stir in the field of networking, and a call for a recommended "best practice" to avoid such problems in the future. Although this is an opportunity that should not be missed, and the paper reports progress toward making the recommendation, there is still much work to be done.

In this paper the application of formal methods is confined to activities associated with designing, validating, and verifying specifications. Activities related to implementation, such as code generation, testing, and maintenance, are not considered.

This paper considers only "lightweight" formal methods, where such methods consist of building a small, abstract formal model of the key concepts of a system, and then analyzing the model with a fully automated ("push-button") tool that works by exhaustive enumeration over a bounded domain of possibilities. In doing so, the tool proves assertions for all model instances within specific size limits. These methods are easier to use and require less specialized knowledge than others, avoiding the objection of "can take years to master."

In this paper lightweight formal methods are represented by Alloy [3] and Spin [4]. Both Alloy and Promela (the modeling language of the model-checker Spin) are richly expressive languages, relative to other modeling languages in their categories. Both the Alloy Analyzer and Spin are mature, well-engineered tools with convenient user interfaces. A recent comparison of the Alloy Analyzer, Spin, and four other push-button tools (CADP, FDR2, NuSMV, and ProB) found Alloy and Spin to perform better than the others [5]. For all these reasons, Alloy and Spin would be leading candidates in most attempts to select a lightweight formal method for a problem.

Because of the tight connection between the expressive power of a modeling language and the algorithms used to analyze it, modeling languages and analyzers tend to come in closely intertwined pairs. The conventional wisdom for selection seems to be "choose the language that best fits the system to be modeled." This paper reports on a project in which the conventional wisdom was useless for selecting one method over another because the system is easily modeled in both Alloy and Promela. The project is described, and its results summarized, in Section 2.

The method employed to compare Alloy and Spin was very simple: attempt to do the same project with both methods, and observe the results. It yielded interesting conclusions because the project was difficult and unstructured, leading to important discoveries in its own right. Such a challenging project pushed both Alloy and Spin to their limits, revealing points of comparison that might not have been evident from easier projects. The points of comparison are organized according to the four stages of the project (modeling, assertions, model exploration, and verification), and refer directly to the labor and feasibility of completing each stage. In contrast, the points of comparison in another recent study [5] refer only to the relative ease of modeling a few classes of properties, defined formally, and have little relationship to overall project success. The comparisons on four project stages are presented in Sections 3 through 6, respectively.

The comparisons are summarized in Section 7. They are, at least, somewhat surprising, which shows that they break new ground. The section reports not only real advantages observed during the project, but also expected advantages that proved to be illusory. It proposes some tentative recommendations for selection of formal methods for lightweight modeling and analysis of network protocols.

One limitation of this research method is that the comparison is based on a single case study. This is not unusual; for example, the recent comparison mentioned above [5] also used only a single case study, and it was a well-known toy example (a library information system) rather than an open research problem.

Another potential limitation of this research method is that the same person applied both formal methods, one after the other. This means that subjective judgments such as "intuitive," "natural," or "easy to use" would be biased. However, this is not a limitation because none of the comparisons are subjective.

The conclusions in Section 8 discuss future work that is needed to solidify recommendations for network protocols. This section also offers some general observations about the formal-method selection problem, as the lessons learned in this area are not confined to the application of formal methods to network protocols.

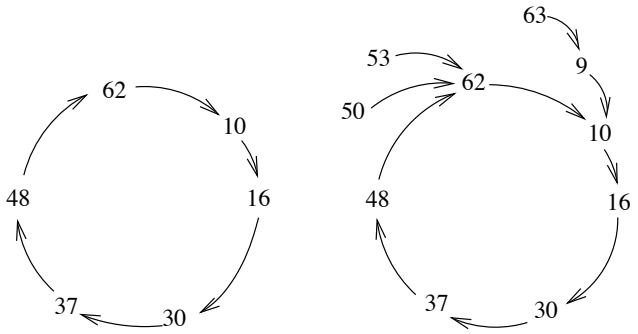

Fig. 1. Ideal (left) and valid (right) networks. Members are represented by their identifiers.

2. The case study: Chord

The distributed hash table Chord was first presented in a 2001 SIGCOMM paper [6]. This paper has been the fourth-most-cited paper in computer science for several years, according to CiteSeer, and won the 2011 SIGCOMM Test-of-Time Award.

A Chord network is structured as a ring. The introductions of both [6] and [7] say of the protocol that maintains the ring structure, "Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance." The papers refer to [8] for the proof of correctness. Invariants of the ring-maintenance protocol are listed in [9].

The reality that was revealed by lightweight modeling and analysis is that, even with simple bugs fixed and optimistic assumptions about atomicity, the Chord ring-maintenance protocol is not correct. Of the seven properties claimed invariant of the protocol, not one is actually an invariant; some (or perhaps all) of the papers analyzing Chord performance are based on misunderstandings of how the protocol works [10]. Eventually I found a version of the protocol that works under reasonable operating assumptions and can be proven correct by formal methods. The remainder of this section gives a brief introduction to Chord.

Every member of a Chord network has an identifier (assumed unique) that is an m-bit hash of its IP address. Every member has a *successor* pointer, always shown as a solid arrow in the figures. Figure 1 shows two Chord networks with $m = 6$, one in the *ideal* state of a ring ordered by identifiers, and the other in the *valid* state of an ordered ring with appendages. In the networks of Figure 1, key-value pairs with keys from 31 through 37 are stored in member 37.

While running the ring-maintenance protocol, a member acquires and updates a *predecessor* pointer, which is always shown as a dotted arrow in the figures. It also acquires a list of extra successors. The *second successor* is always shown as a dashed arrow.

The ring-maintenance protocol is specified in terms of operations, each of which changes the state of at most one member. In executing an operation, the member often queries another member, then updates its own pointers if necessary. Some operations sometimes entail a second query before completion. The specification of Chord assumes that inter-node communication is reliable, so we are not concerned with Chord behavior when inter-node communication fails.

A node becomes a member in a *join* operation. A member node is also referred to as *live*. When a member joins, it contacts an existing member and gets its own correct current successor from that member. The first stage of Figure 2 shows successor and predecessor pointers in a section of a network where 10 has just joined. Node 10 got its successor 19 from node 7, which also has 19 as its successor.

When a member *stabilizes*, it learns its successor's predecessor. It adopts the predecessor as its new successor, provided that the predecessor is closer in identifier order than its current successor. Members schedule their own stabilize operations periodically.

Between the first and second stages of Figure 2, 10 stabilizes. Because its successor's predecessor is 7, which is not a better successor for 10 than its current 19, this operation does not result in a state change and is not shown.

After stabilizing (regardless of the result), a node notifies its successor of its identity. Thus a stabilize operation always causes a *notified* operation. The notified member adopts the notifying member as its new

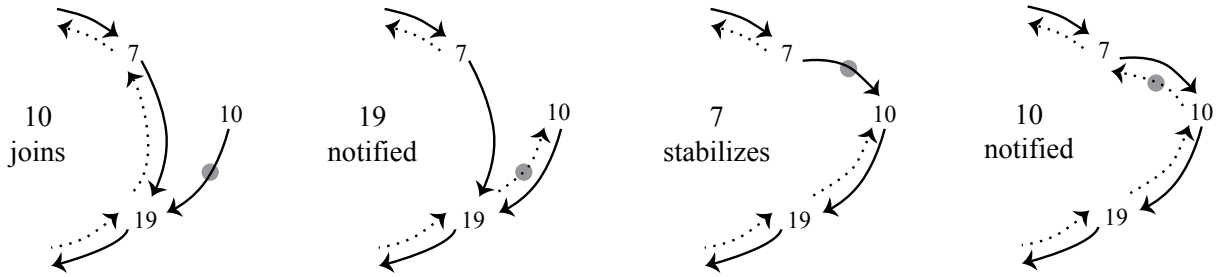

Fig. 2. A new member becomes part of the ring. After each operation, a gray circle marks the updated pointer.

predecessor if the notifying member is closer in identifier order than its current predecessor. In the second stage of Figure 2, 10 has notified 19, and 19 has adopted 10 as its new predecessor.

In the third stage of Figure 2, 7 stabilizes, which causes it to adopt 10 as its new successor. In the last stage 7 notifies 10, so the predecessor of 10 becomes 7. Now the new member 10 is completely incorporated into the ring, and all successor and predecessor pointers are correct.

The assumption of the protocol is that a member in good standing always responds to queries in a timely fashion. A node ceases to become a member in a *fail* event, which can represent failure of the machine, or the node's silently leaving the network. A member that has failed is also referred to as *dead*. After a member fails, it no longer responds to queries from other members. With these assumptions, members can detect the failure of other members perfectly by noticing whether they respond to a query before a timeout occurs. Another assumption about failure behavior is that successor lists are long enough, and failures are infrequent enough, to ensure that a member is never left with no live successor in its list.

Failures can produce gaps in the ring. These disruptions are repaired with the help of *reconcile, update,* and *flush* operations, each of which is executed periodically by each member, according to its own schedule.

When a member *reconciles*, it adopts its successor's successor as its second successor (if successor lists are longer than two, it adopts its successor's entire successor list except for the last entry, for which it has no room). When a member *updates*, it replaces a successor pointer to a dead member by the first successor pointer in its list that points to a live member. When a member *flushes*, it discards a dead predecessor.

It is well known that a Chord network must preserve the following structural property (which will be formalized as *valid* in Section 4). Defining a member's *best successor* as its first successor pointing to a live node (member):

- there must be a ring of best successors;
- there must be no more than one ring;
- on the ring of best successors, the nodes must be in identifier order;
- from each member in an appendage, the ring must be reachable through best successors.

If any of these rules is violated, there is a disruption in the structure that the ring-maintenance protocol cannot repair. Because of the way the lookup protocol works, the inevitable result is that some members will not be reachable from some other members.

Figure 3 shows one way that the *valid* property can be violated. Only successors and second successors are shown, and the captions between stages describe sequences of several operations. The figure actually represents a family of counterexamples, one for every odd ring size greater than one. Using similar scenarios, there is also a counterexample for every even ring size greater than one. With even ring sizes, rather than becoming disordered, the ring breaks into two disconnected rings.

A network is *ideal* when all its pointers are correct. The correctness criterion for the ring-maintenance protocol is simple: *In any execution state, if there are no subsequent join or fail events, then eventually the network will become ideal and remain ideal.* This is not a particularly stringent requirement, as it allows the protocol ample time and no further disruptions while it works to repair the ring.

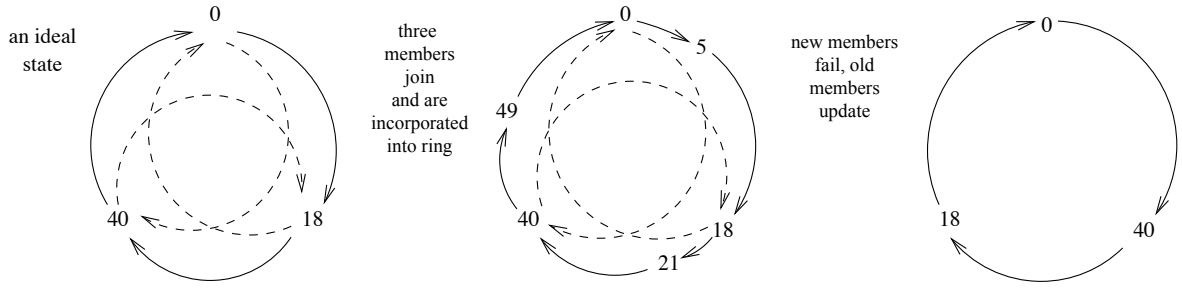

Fig. 3. Three stages (left to right) creating a ring whose nodes are not in identifier order.

3. Modeling

The first step in studying Chord is to build a lightweight model of how it works. This requires making a few decisions about how to formalize it in a succinct but sufficiently faithful way.

As described in Section 2, members communicate using a reliable query/response message pair; if there is no response after some specified period of time, the querying member knows that the queried member is dead or gone. This is so simple that it can be modeled with shared memory, abstracting away the network completely.

An operation that requires one query to another node can be modeled as a single atomic event; this event can be thought of as occurring at the instant when the queried node responds to the query. Although the querying member will not change its state until some time later, as long as it responds to no queries while it is waiting for a response, the discrepancy is unobservable. An operation that requires two queries must be modeled as two atomic events.

To add more necessary detail to this explanation, if a member is waiting for a response and receives a query, it should actually send an interim "will respond soon" message. Otherwise, if the response to the first query takes too long or does not come, the second query might time out erroneously.[1]

Lightweight analysis will require a limit on the number of nodes (potential members) analyzed, although this number does not much affect the complexity of the model, *i.e.,* the length of the program to be written. It is also necessary to decide on the length of each member's successor list, which may very well affect the complexity of the model, as longer lists may require more data manipulation. For simplicity, the chosen length is 2.

In a real Chord implementation, successor lists must be long enough so that the probability that a member's successor list contains no live successors is very low. For lightweight modeling this probabilistic assumption must be made deterministic. Specifically, the model must be constrained so that a member cannot fail if it would leave some other member with no live successor.

It is interesting to compare Promela and Alloy, because they are so very different. Here is a brief summary of each language, arranged for point-by-point comparison.
Promela:

1. There are concurrent processes, communicating through shared global data. In addition to variables, the data structures are messages with multiple fields, bounded queues, and fixed-size arrays. Network communication is simulated by enqueueing and dequeueing messages. In a process program, there are control structures offering sequence, choice, and iteration over guarded commands.
2. As in most programming languages, time is implicit.
3. A model can be executed or analyzed (model-checked). During execution or analysis, a trace is generated by running all the processes concurrently. More specifically, each trace is an arbitrary interleaving of enabled execution steps from all the concurrent processes.
4. There are two sources of nondeterminism. Within a process program, the choice of guarded command is often nondeterministic. Also, during execution or analysis, the interleaving of events in different processes is nondeterministic.

[1] The Chord papers simply assume that all operations are atomic, whether they require no, one, or two queries. They do not discuss any of these details, leaving them for implementers to discover on their own.

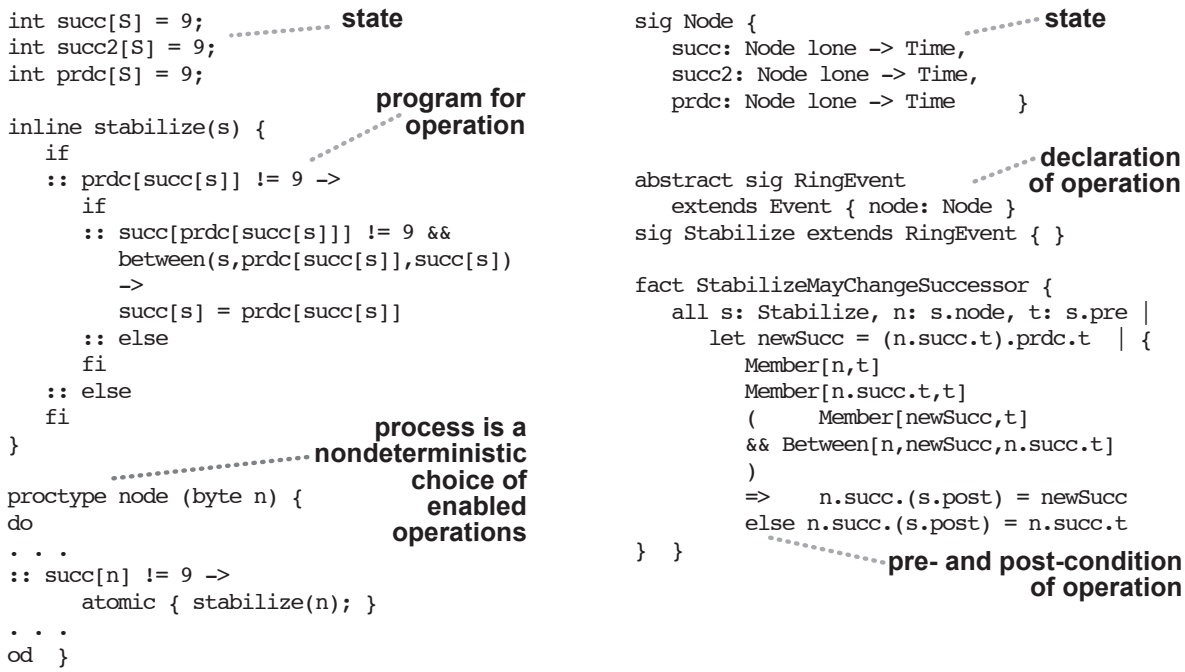

Fig. 4. Annotated Promela (left) and Alloy (right) fragments of the Chord models.

5. The size of the model, for example the number of nodes in a Chord network, is part of the Promela program.

Alloy:

1. The model state consists of sets and relations over individuals. Properties of the state are expressed in a rich language combining relational algebra, first-order predicate logic, transitive closure, and objects.
2. Time must be made explicit. To do this, one represents timestamps as individuals in a totally ordered set. The time-varying state of an object is represented using a relation whose tuples include a timestamp. Events are also individuals, and a trace can be regarded as a sequence of alternating events and timestamps, where the order on the timestamps enforces an order on the events. The precondition of an event is a fact about all the tuples with the pre-time of the event as their timestamp. The postcondition of an event is a fact about all the tuples with the post-time of the event as their timestamp.
3. During analysis, a trace is specified by constraining its event sequence, *e.g.,* "a Stabilize followed by a Notified."
4. There is one source of nondeterminism, arising simply from logical underconstraint. Multiple instances (each a specific value of all the sets and relations) can exemplify or satisfy the given properties.
5. Most size bounds are specified only for analysis, and are not part of an Alloy model. For example, the Alloy model of Chord does not mention the number of nodes. The length of successor lists does appear in the model, however.

Despite these radical differences, it is straightforward to model the Chord ring-maintenance protocol in either language, although it is important in each case to know the common language idioms. For Chord in Promela, the most important language features are inline macros and atomic sequences [4]. For Chord in Alloy, the most important idioms are time as a column of a relation and frame conditions [3]. Figure 4 shows some annotated fragments of the Promela and Alloy models. The figure is intended to give a general impression, and is not accurate in every detail.[2]

[2] All of the models referred to in this paper are available on the Web at `www2.research.att.com/~pamela/model.html`.

There is one difficulty with modeling Chord in Promela, but it is more easily explained as part of the next section.

4. Assertions

The second step in studying Chord is to formalize the properties it is required or expected to satisfy. The formalizations of safety and progress properties are quite different, so we will look at them separately.

The Alloy assertion language is the same as its modeling language. The Chord safety properties are invariants on the network structure. As explained in Section 2, a necesssary property is that a network be *valid* at all times, where the predicate *valid* is defined in Alloy as:

```
pred Valid [t: Time] {
  let members = { n: Node | some n.succ.t } |
  let ringMembers = { n: members | n in n.^(bestSucc.t) } |

     {  some ringMembers                          -- at least one ring

        all disj n1, n2: ringMembers |
           n1 in n2.^(bestSucc.t)                  -- at most one ring

        all disj n1, n2, n3: ringMembers |
           n2 = n1.bestSucc.t => ! Between[n1,n3,n2]  -- ordered ring

        all na: members - ringMembers | some nc: ringMembers |
           nc in na.^(bestSucc.t)             -- connected appendages
     }
}
```

This definition relies on definitions of *bestSucc*, which is a member's first live successor, and *between*, which tests the ordering of identifiers. It is concise and very readable, at least for those familiar with Alloy syntax. This pleasantness is to be expected, as *valid* is a graph property, and a graph is a kind of relation. The following Alloy code asserts the safety property that join operations preserve validity.

```
assert JoinPreservesValidity { some Join && Valid[trace/first] => Valid[trace/last] }
```

This assertion is intended to be checked over instances with one event (which must be a join event because of the *some* predicate) and two timestamps. It says that if the state represented by all the tuples with the first timestamp (the pre-state of the join event) is valid, then the state represented by all the tuples with the second timestamp (the post-state of the join event) is also valid.

In Promela a safety assertion can be inserted at any point in the code. For example, an assertion that the state is *valid* would be inserted immediately after any state change. The problem is that the assertion language consists of Boolean expressions over the state variables; it is not possible to express the often-complex graph properties needed for analyzing Chord.

To express the safety assertions, it is necessary to program checks for the graph properties in C and to call the C code from Spin. For example, *valid* is a C function that checks the model state for validity, and it is invoked with the following Promela statement:

```
assert c_expr{ valid(now.succ,now.succ2,now.prdc) }
```

The *now* keyword tells Spin to use the array values from the current model state.

This is a grave disadvantage. The time required to learn basic C and how to call it properly from Spin is approximately the same as the time required to learn Promela and the basic use of Spin, making the overall startup time approximately twice that required for Alloy. Also, it is obviously more difficult to program a graph property in C than to state it declaratively in a relational language.

This disadvantage applies not only to formalizing assertions, but may also apply to modeling the protocol itself. The modeling difficulty mentioned at the end of Section 3 concerns the predicate governing whether a member can fail (recall that a member cannot fail if it would leave some other member with no live successor). Like the safety properties, this is a graph property, and must be programmed in C.[3]

[3] Promela inline macros work at the statement level and not the expression level. For this reason, *between* on the left side of Figure 4 is actually C code.

The correctness property for Chord is a progress property (see Section 2). It can be expressed in linear-time temporal logic, using the temporal progress operator $\Diamond$ (*eventually*). Let *churnStopped* be a Boolean history variable that is initialized to false, and is set to true nondeterministically. Once it is true, it remains true, and all joins and failures are disabled. Using this history variable, the correctness property for Chord is:

```
( <> churnStopped ) -> ( <> [] ideal )
```

where `<>` is $\Diamond$ and `[]` is the temporal operator $\Box$ or *always*. Informally, in any execution trace in which churn eventually stops, the network eventually becomes ideal and stays ideal.

In principle, Spin can check this progress property (see Section 6 for further information). Mathematically a progress property can only be falsified by an infinite trace. Model-checkers can find finite counterexamples, however. A counterexample is a finite trace with a loop (first and last states the same), where the loop does not contain a goal state. This shows that the system can loop forever without reaching its goal.

An immediate difficulty is that Chord is a "busy waiting" protocol, in the sense that each member checks for updates periodically, without knowing ahead of time whether an update will be required or not. An ineffective update check is itself a loop that may not contain a goal state, falsifying the progress property. Spin gives us two ways to deal with this difficulty. One way is to specify model-checking with weak fairness. The other way is to use an additional history variable in a more complex temporal-logic assertion that excludes traces with busy waiting from consideration.

There are no temporal operators in Alloy. Strictly speaking the progress property could be expressed in Alloy using quantification over timestamps, but there is no point in doing so because the Alloy Analyzer could not check it meaningfully (see Section 5). For all practical purposes, progress properties cannot be asserted in Alloy.

5. Model exploration

The third step in studying Chord is to use analysis to debug the model and assertions, and to check which assertions are true. This entails producing examples of desirable behavior and counterexamples to conjectured assertions. Model exploration is by far the most important step, because it is the step that takes most of the user's time.

Producing examples and counterexamples is done somewhat differently in the two tools, but is straightforward in both. With the Alloy Analyzer, to produce an example, you *run* a *predicate*. To search for counterexamples, you *check* an *assertion*. There is no difference but the keywords.

With Spin, to search for counterexamples to all safety assertions in the Promela code you simply run the model-checker, and it will report a counterexample if it finds one. To produce an example, you write an assertion that the desired state has been reached, and insert its negation in an *assert* statement where the assertion might become true. When the model-checker runs and the desired state is reached, Spin will consider it an error and report the trace.

Understanding a protocol whose behavior is as complex and unpredictable as Chord requires studying many, many odd example traces [10]. The easier it is to see what is going on in a trace, the easier the overall job will be.

With Spin, my C code printed snapshots of the network structure in the form of arrays of pointers. The Alloy Analyzer has excellent visualization tools for customized display of examples and counterexamples in the form of graphs. So it was expected that the Alloy Analyzer would prove superior to Spin for visualization,[4] but this was not the case.

Figure 5 is the smallest counterexample to correctness of the original Chord protocol. The Alloy Analyzer approximates this picture, but unfortunately not well enough to comprehend more complex counterexamples. There is a feature in the Analyzer that operates on model instances with timestamps, producing a time-sequence of snapshots as in Figure 5. However, the user interface of this feature shows only one snapshot at a time. Also, as with almost all graph-layout programs, the Analyzer displays each graph to optimize certain

[4] The Spin graphical user interface iSpin incorporates visualization of message sequence charts. For many projects this is extremely useful, but the Chord models use shared-memory communication rather than messages, and it is the state graph that matters. Also, iSpin cannot be used when embedded C code is present.

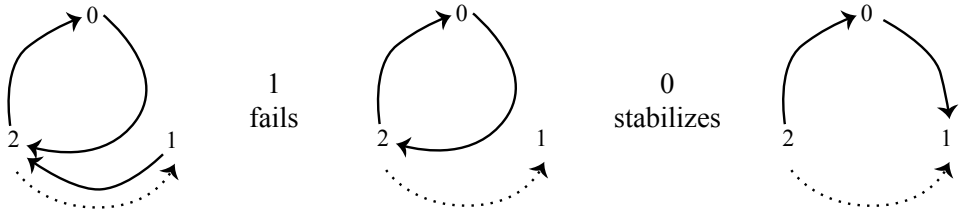

Fig. 5. Three stages (left to right) creating a broken ring that cannot be repaired by the Chord protocol.

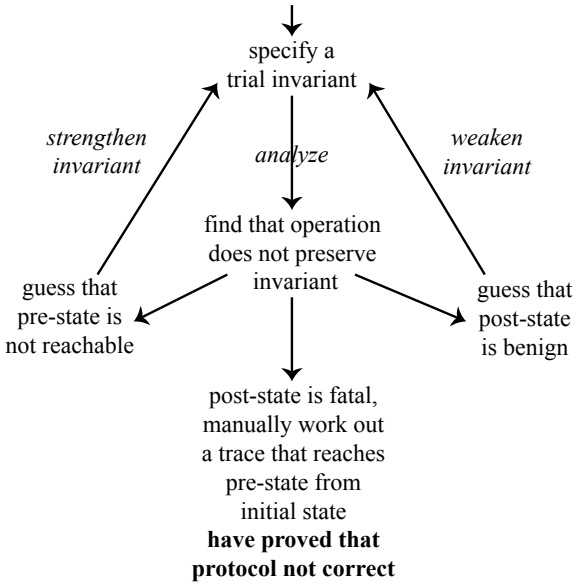

Fig. 6. The process of investigating Chord with Alloy.

layout metrics, which means that the nodes move from snapshot to snapshot. Understanding Chord requires a fixed node layout in which nodes are arranged in a circle in identifier order.

Whether using Spin or the Alloy Analyzer, I ended up drawing a picture like Figure 5 by hand for each example or counterexample.

Model-checking and Alloy analysis are fundamentally different. As it generates all loop-free traces, Spin creates an explicit internal representation of the entire reachable state space. Thus the notion of a time-sequence of computational steps is built into model-checkers, and they are optimized for it. The representation of the reachable state space is typically large and certainly not human-readable.

In Alloy neither timestamps nor events are different from any other type of individual. There is no built-in optimization for the passage of time, and analysis can cover only very short traces, for example traces with up to 3 or 4 events in the case of Chord.

Consequently, to use Alloy to analyze Chord, it is absolutely necessary to have an explicit global state invariant written as an Alloy predicate. This global invariant is a concise, human-readable description of the reachable state space. With it, the Analyzer can be constrained to start only from reachable states, and enumerate what can happen in the next few computational steps after them.

The effect of this theoretical difference on practice is profound. Figure 6 is an informal flowchart of the process of investigating, with the Analyzer, the original version of Chord and some variants of it. I began this process with the assumption that Chord was correct and therefore must have a global invariant at least as strong as *valid*, as the creators of Chord claimed to have proved it correct.

The process described in Figure 6 quickly found some easily fixed bugs, such as the one shown in Figure 5. After that the process was arduous and frustrating. To make it easier, every operation was assumed to be atomic. Eventually it was possible to find several post-states that are not *valid* (a fatal error) and to work

out manually, for each, a trace that reaches it from the initial state. Each such counterexample is a proof that the protocol is not correct, even with simple bugs fixed and optimistic assumptions about atomicity.

The longest such counterexample trace has 21 events. It is easy to check that it really is a trace with the Analyzer, even though it was generated manually, because little searching is required for the Analyzer to "find" a well-specified trace.

Studying all the counterexamples to correctness of the original protocol [10], it became possible to assemble a "best assembly" version of Chord by choosing the best pieces of pseudocode and textual hints from three different Chord papers, fixing other small bugs, and making optimistic assumptions about atomicity. It was not possible to tell whether the "best assembly" version is correct using Alloy, because the process of Figure 6 did not converge to either a useful global invariant or a counterexample.

At this point it was necessary to start using Promela and Spin. Longer traces from Spin revealed that the "best assembly" version is not correct either. A typical example of these traces is the equivalent of about 50 events modeled in Alloy.

6. Verification

After more experimentation with various strategies, it was finally possible to find a new version of Chord that is correct and is implementable with realistic assumptions. It was also possible to find an Alloy global invariant for it, so that it can be analyzed with both tools.

The fourth and final step in studying Chord is to produce a convincing proof that the new version is correct. The form of this proof is completely conventional. What makes it interesting is the comparative usefulness of Spin and the Alloy Analyzer.

The proof has three steps:

1. Show that *valid* is an invariant.
2. For those operations that consist of two atomic events, show that the postcondition of the first implies the precondition of the second, even if other events intervene.
3. Prove the progress property
   $(\Diamond \mathit{churnStopped}) \implies (\Diamond \Box \mathit{ideal})$.

We begin with showing that *valid* is an invariant, by means of automated analysis. Because we can only analyze networks up to some size limit, this is not a mathematical proof in the purest sense, but we do expect to achieve a very high level of confidence.

Chord is an easy problem in this respect, because ring structures have a great deal of symmetry. For example, to verify assertions relating pairs of nodes in a ring structure, it is only necessary to check rings of up to size 4 [11]. This result is not directly relevant to Chord, because Chord's assertions are global, but it does show the strength and significance of ring symmetry.

In considering the experimental results for Chord, note that the *ring size* is defined as the number of members in the ring. The *node size* is the number of nodes analyzed, which may be larger than the ring size because some members are in appendages and some nodes are not members at all.

The original version of Chord has a minimum ring size of 1. Concerning counterexamples found during model exploration:

- Many new counterexamples were found with node sizes 2, 3, and 4:
- One new counterexample was found with node size 5.
- No new counterexamples were found with larger sizes.

It turns out that many Chord problems occur when a node's pointers wrap around, for example when a node's successor or second successor is itself. One of the strategies used to get a correct version of Chord is to require a minimum ring size one greater than the length of the successor list, so that no pointer ever wraps around. This means that the model of the correct version has a minimum ring size of 3. Concerning counterexamples found during exploration of various versions with a minimum ring size of 3:

- Many new counterexamples were found with node sizes 4 and 5.
- One new counterexample was found with node size 6.
- No new counterexamples were found with larger sizes.

-w	memory in megabytes	elapsed time **Version 1**	hash factor **Version 1**	elapsed time **Version 2**	hash factor **Version 2**
31	256	3.14 hr	1.754	3.78 hr	2.336
32	512	6.17 hr	1.774	1.76 hr	9.087
33	1024	12.4 hr	1.787	9.89 hr	3.718
34	2048	23.9 hr	1.798	9.64 hr	7.518
35	4096	48.1 hr	1.869	29.7 hr	11.51
39	65,536	185 hr	3.640	28.6 hr	193.8

Table 1. Spin safety checks in supertrace mode.

This is empirical evidence for the "small scope hypothesis" [3], which states that most bugs have small counterexamples.

With the Alloy Analyzer, the invariant can be checked quickly for node size 8. As this seems adequate to achieve high confidence in the result, no attempts were made to check larger sizes, and Step (1) of the proof can be considered complete. Completing Step (2) of the proof is similar to checking the invariant.

With Spin model-checking, the node size is a problem. Models with node size 4 can be checked exhaustively. Checking of a model with node size 5 aborted after using over 300 gigabytes of memory. Yet the experiences with counterexample size recounted above indicate that node size 6 is the bare minimum for credible analysis of a version with a minimum ring size of 3.

It is interesting to note that the previously mentioned comparative study [5], analyzing a model of a library system, got almost exactly the same quantitative results. At most, Spin could analyze models with 5 instances of the critical object type. The Alloy Analyzer, on the other hand, could handle 8 easily.

Another dimension in which model-checking can be limited is the length of traces checked. For purposes of comparison, one event in the Alloy model corresponds to about 10 execution steps in the Promela model. However, checking all Alloy traces of length 4 is not equivalent to checking all Promela traces of length 40. The traces in the set checked by Alloy would begin with all states that satisfy the global invariant. The traces in the set checked by Spin would begin with a single initial state. Hence the set of traces checked by Alloy would be much larger.

The longest counterexample trace found by Spin was 600 steps. Somewhat arbitrarily, we can choose 1000 steps as a convincing trace length, and limit model-checking to loop-free traces of that length or less. This is a huge simplification, as analysis of the model with node size 5 found traces with lengths of 1.4 billion.

Another dimension in which Spin model-checking can be limited is the memory used to represent the reachable state space. With Spin's "bitstate" or "supertrace" mode, a fixed-size hash table is used to represent the states reached so far. If two real states collide in the hash table, then model-checking cannot distinguish them. The "hash factor" is a statistical measure of how well a supertrace check covers the real state space. If the hash factor is over 100, there is high confidence that coverage is exhaustive or nearly so. If the hash factor is nearly 1, there is near certainty that only a very small fraction of the true state space was visited in the run. Note that the hash factor is independent of the trace length, in the sense that the hash factor refers only to the state space reachable within the given trace length, not to the entire reachable state space.

Spin analysis of the correct version of Chord, to check the safety assertion *valid*, with node size 6, with trace length limit 1000, and in supertrace mode, yields the results in Table 1. The *-w* option determines the size of the hash table. Version 1 is the version of the Promela model I wrote. Version 2 is similar, but has benefited from some expert tuning.

Although the last analysis in the table reported a good hash factor, the results from analyzing Version 2 are marred by the absence of a consistent progression, as seen in the results from analyzing Version 1. Doubts are introduced by the absence of a consistent progression, by the fact that trace length is limited to the equivalent of 100 events, and by the fact that the node size is the minimum plausible number of 6. Together these doubts mean that a high level of confidence is not achieved, so it seems that Spin cannot be used for a convincing proof of Chord safety properties. Although understandable in retrospect, it was initially a surprise that the presence of a global invariant gives Alloy such an important performance advantage.

Alloy verification has another advantage over Spin verification that is independent of analytic performance. The Alloy proof is comprehensible to humans, which means that it could be converted to a mathematical proof for networks of any size. Even if there were a Spin verification for examples of convincing size,

	Promela + C / Spin	**Alloy / Alloy Analyzer**
Real Advantages	not necessary to know a sufficiently strong global invariant (5) supports progress assertions (4)	half the startup time (4) safety assertions are declarative rather than procedural (4) can be used for a convincing proof of correctness (6)
Illusory Advantages	the better choice for all network protocols (3) automated verification of progress assertions (6)	automated visualization of examples as graphs (5)

Table 2. Summary of comparative advantages on the Chord case study.

it would be based on an unreadable internal representation of the reachable state space, and could not be converted to a mathematical proof.

Of the three proof steps enumerated at the beginning of this section, both Steps (1) and (2) require verification of safety properties, so the same Spin deficiencies apply to both. Step (3) requires different techniques, as we are switching from safety to progress.

The proof of the progress property for the Alloy model of the correct version is partly manual. It is straightforward (and illuminating) to define a natural number that measures the error in the pointers of a Chord network, to show that an *ideal* network has error 0, and to show that every effective repair operation (one that updates a pointer) reduces the error. The proof is completed by checking two lemmas with the Analyzer, both of which are safety properties. One lemma says that if the network is *valid* but not *ideal*, some effective repair operation is enabled. This ensures that eventually an effective operation will reduce the error. Because the network is finite, a finite number of reductions will make the error 0. The second lemma says that if the network is *ideal*, no effective repair operation is enabled. This ensures that no Chord operation changes the pointers of an *ideal* network.

In Spin, checking progress assertions takes approximately twice the resources of checking safety assertions. So, although Spin can be used to debug with progress assertions—which can be extremely useful in its own right—it cannot be used to verify the Chord progress property.

7. Comparisons

Table 2 summarizes the results from the previous four sections, in terms of real, significant advantages on either side. Each advantage is marked with the section or sections in which it is discussed.

It is important to stress that the table includes only *comparative* advantages of Spin and Alloy in relation to each other. The advantages of using either Spin or Alloy, in comparison to doing no modeling at all, are far more extensive than the advantages of either over the other.

For completeness, the table also includes illusory advantages. These are the advantages I would have expected from general knowledge of Alloy and Spin, but that turned out to be wrong or irrelevant in this case. As discussed in Section 3, Alloy is as good for making a shared-memory model of Chord as Promela is. Even though model checking is usually associated with network protocols and Alloy is not, Promela is not necessarily a better choice than Alloy for modeling protocols. As discussed in Section 6, even though Spin supports the assertion of progress properties and their use for debugging, automated verification of progress properties was not feasible in this project because of computational complexity. As discussed in Section 5, despite automated visualization in the Alloy Analyzer, it was necessary to hand-draw pictures from the outputs of both Spin and Alloy. This problem is unlikely to persist, however, as the automated visualization is frequently improved.

If it were not for the absence of a sufficiently strong global invariant (until a provably correct version was finally found), Alloy would be the clear winner in this particular comparison. Its strength in handling graph properties is a good match for the complexity of the problem. This is true for progress as well as safety, because the heart of the progress proof is a monotonically decreasing termination function on the network graph. Equally important, the Alloy Analyzer can analyze scopes large enough to provide convincing evidence.

The biggest problem in carrying out this case study was that the Chord protocol was designed—*and claimed to be provably correct*—without a known global invariant. This is the reason it was necessary to resort to Spin. It may be that the only real solution to this problem is to educate people on the importance of invariants as a design tool. Even so, it would help to accompany exhortation with guidance on how to design invariants that are abstract enough to be simple yet concrete enough to be implementable, or at least implementable with high probability.

This case study, along with other projects, suggests a possible generalization about network protocols. In networking so-called "control plane" protocols, most prominently routing protocols, tend to have the purpose of computing and maintaining distributed representations of networks as graphs. The Chord ring-maintenance protocol resembles a control-plane protocol, and conclusions about formal methods for Chord may be applicable to other control-plane protocols. In other work, algebraic techniques have been used successfully to formalize generic properties of routing protocols [12], and Alloy has been used successfully to formalize properties of the Border Gateway Protocol (BGP) [13].

In contrast to control-plane protocols, networking also uses many end-to-end protocols, enabling endpoints to synchronize and communicate with each other. Some of these protocols are defined for specific applications, while others offer transport services to a variety of applications. In these protocols there is no graph-like complexity, as a small number of actors (usually only two) is involved. On the other hand, there can be a great deal of complexity in the control states and event sequences required.

For end-to-end protocols Spin is likely to be a better choice than Alloy, because it is probably easier to describe complex event sequences and their control states in Promela than in Alloy. The issue of startup time in Table 2 does not apply, because it is not necessary to augment Promela with C. The complex end-to-end application protocol Session Initiation Protocol (SIP) has been successfully modeled with Promela [14], and Spin performance has been adequate to provide convincing evidence of the correctness of SIP-based network elements [15]. Note that these experiences are not as reliable as the Chord comparison, however, because no attempt was made to model or verify SIP with Alloy.

In other work, the well-known transport protocol TCP has been modeled in HOL [16]. TCP is complex because it has many aspects, including data handling, buffering, congestion control, and timing. The TCP model is different from any other model mentioned here because it is not lightweight. Rather, it is the result of a massive effort to provide a complete formal specification of all aspects of this important protocol.

8. Conclusions

The original version of Chord was claimed to be correct on the basis of an unsuccessful attempt at an informal proof (in [8]). This shows that the standards of specification and verification in some areas of computer-science research are far behind what is technically feasible.

My original goal for this project was to make an enthusiastic recommendation of some lightweight method as a "best practice" for modeling and analyzing all tricky network protocols, or at least for protocols similar to Chord. The surprising result that Chord is not correct creates an opportunity for publicizing the value of lightweight modeling and analysis. Although the ultimate goal has not been reached, some progress has been made, as represented by the tentative recommendations in Section 7.

To make a stronger recommendation, it would be necessary to offer potential users more help with global invariants that describe the reachable state space of a model. (Of course, the ideal tool would discover them automatically, but this is rather difficult!) Work on Chord provides a wealth of experience that might be used to learn more about what useful invariants for these protocols look like, and how they might be constructed. For example:

- Maintenance of the ring structure is only part of the correctness of a distributed hash table (DHT). There are other properties that are valuable for its actual purpose, including *lookup consistency*, meaning that if a key-value pair is in the table, a lookup always finds it, and *key consistency*, meaning that all members

agree about which member is responsible for a particular key [17]. These properties do not seem to be required of the DHTs in common use, and may be expensive to satisfy in their strongest form. But if they were requirements, even in some weakened form, many versions of Chord would be rejected immediately, and the remaining ones would have stronger invariants.

- Some Internet behaviors have a significant effect on Chord [18]. This work on the underlying communication network could be consulted to determine which infrastructure properties can or must be assumed.
- To implement a DHT efficiently, it is almost certainly necessary to make assumptions that are not true, but that do hold almost all of the time. It would be extremely valuable to all protocol designers to learn more about probabilistic assumptions and how to make them wisely.

Although this comparison focuses on specification rather than implementation, it is worth noting that three papers have reported on the application of model checking to implementations of Chord [19, 20, 21]. Because implementations are so much more complex than abstract models, analysis is necessarily incomplete, and the techniques are intended to find bugs rather than attempt or approximate verification. Although all three papers report finding bugs in Chord implementations, none of them found any of the specification bugs described here.

What are the lessons learned concerning research on selection of formal methods? It seems very clear that in-depth case studies applying multiple formal methods to the same real problems are better than superficial studies of toy problems, or studies applying only a single formal method to each problem. Unfortunately, like the obvious need for a tool that discovers invariants automatically, this lesson is difficult to put into practice.

Section 1 referred to the conventional wisdom about selecting the language that best fits the system to be modeled, rejecting it as useless for choosing between Alloy and Promela to model Chord. It turns out that the conventional wisdom is useful, if we include the properties to be verified as well as the functional specification; for Chord, this would emphasize the difficulties of augmenting Promela with C to use Spin. Consequently, one lesson learned is that writing the functional specification is only part of the specification work, and not necessarily the most difficult part.

Tools for lightweight analysis work by exhaustive enumeration over a bounded domain of objects. It now seems that it might be possible to experiment carefully with the relationship between these bounds or scopes and tool performance, and possibly observe some patterns that apply to multiple models. It would be valuable to predict the relationship between scope and tool performance, because it often affects whether a convincing proof of correctness is possible or not.

Acknowledgments

Helpful discussions with Ernie Cohen, Patrick Cousot, Gerard Holzmann, Daniel Jackson, Arvind Krishnamurthy, Gary Leavens, Pete Manolios, Annabelle McIver, Jay Misra, Andreas Podelski, Emina Torlak, Natarajan Shankar, and Jim Woodcock have contributed greatly to this work.

References

[1] S. Keshav. Editor's message: Modeling. *ACM SIGCOMM Computer Communication Review*, 42(3):3, July 2012.

[2] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), October 2009.

[3] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006, 2012.

[4] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.

[5] Marc Frappier, Benoît Fraiken, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In *Formal Methods and Software Engineering*, pages 581–596. Springer-Verlag LNCS 6447, 2010.

[6] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM.* ACM, August 2001.

[7] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1), February 2003.

[8] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. MIT LCS Technical Report 819, `www.pdos.lcs.mit.edu/chord/papers/chord-tn`, 2001.

[9] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 233–242. ACM, 2002.

[10] Pamela Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2), April 2012.

[11] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 85–94. ACM, 1995.

[12] Timothy G. Griffin and João Luís Sobrinho. Metarouting. In *Proceedings of SIGCOMM*. ACM, August 2005.

[13] Matvey Arye, Rob Harrison, Richard Wang, Pamela Zave, and Jennifer Rexford. Toward a lightweight model of BGP safety. In *Proceedings of the 1st International Workshop on Rigorous Protocol Engineering*. Vancouver, British Columbia, October 2011.

[14] Pamela Zave. Understanding SIP through model-checking. In *Proceedings of the 2nd International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 256–279. Springer-Verlag LNCS 5310, 2008.

[15] Pamela Zave, Gregory W. Bond, Eric Cheung, and Thomas M. Smith. Abstractions for programming SIP back-to-back user agents. In *Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications*. ACM SIGCOMM, 2009.

[16] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP and sockets. In *Proceedings of SIGCOMM '05*. ACM, August 2005.

[17] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Tom Anderson. Scalable consistency in Scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, October 2011.

[18] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and DHTs. In *Proceedings of the 2nd Conference on Real, Large, Distributed Systems*, pages 55–60. USENIX, 2005.

[19] Charles Killian, James A. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Symposium on Networked System Design and Implementation*, pages 243–256, 2007.

[20] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, April 2009.

[21] Maysam Yabandeh, Abishek Anand, Marco Canini, and Dejan Kostić. Almost-invariants: From bugs in distributed systems to invariants. Technical report, EPFL NSL-REPORT-2009-007, 2009.