A Practical Comparison of Alloy and Spin

Pamela Zave
AT&T Laboratories—Research
Florham Park, New Jersey USA
Email: pamela@research.att.com

Abstract—Because potential users have to choose a formal method before they can start using one, evidence suggests that research on assessing and explaining 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 conclusions contradict conventional wisdom and expectations on several points. Although the scope of the comparison is narrow with respect to the systems being modeled, it is broad with respect to the range of activities and goals involved in using formal methods.

I. 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 (e.g., transportation, financial, health care) and techniques used (e.g., specification/modeling, proofs, model checking). The survey found only a few mild correlations between the type of software (e.g., consumer electronics, transaction processing) and the techniques used. These results will not help a potential user choose a specific formal method to apply to a specific problem.

The goal of this paper is to shed a small amount of light on selection of formal methods. It might also help to improve the usefulness of formal methods by pointing out weaknesses, and to encourage development of more sophisticated methods of comparison and selection.

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. Either one is a formal method that can be recommended in general without reservation.

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 because the system is easily modeled in both Alloy and Promela. The project is described, and its results summarized, in Section II.

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 deep and difficult, and led to research discoveries in its own right. Such a difficult 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. These comparisons are presented in Sections III through VI, respectively.

Spin falls into the category of discrete model checkers, as opposed to hybrid ones that combine discrete and continuous analysis. No well-known language is similar to Alloy, because of its analyzability rather than because of the language itself.
The comparisons are summarized in Section VII. 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.

One limitation of this research method is that the scope of the comparison is very narrow with respect to the class of systems to be modeled. On the other hand, we need to accept the reality that we know little about selection of formal methods. Because we know so little, we do not yet know which aspects of the modeled system are safe to generalize or ignore. From this perspective, a comparison of formal methods on one project is the only valid comparison. The conclusions (Section VIII) discuss how the scope might be broadened in this dimension.

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.

In addition to suggestions for improving methods of comparison and selection, Section VIII also discuss the implications of this work for lightweight formal methods, their adoption, and their future improvement.

II. THE CASE STUDY: CHORD

The distributed hash table Chord was first presented in a 2001 SIGCOMM paper [5]. This paper is the fourth-most-cited paper in computer science,\(^2\) and won the 2011 SIGCOMM Test-of-Time Award.

A Chord network is structured as a ring. The introductions of both [5] and [6] 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 [7] for the proof of correctness. Invariants of the ring-maintenance protocol are listed in [8].

The reality revealed in this project 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 [9]. 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.

\(^2\)And has been for several years, according to CiteSeer.

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 machine becomes a member in a join operation. When a member joins, it contacts an existing member and gets its own correct current successor from that member. Joins cause the ring to have appendages such as those on the right side of Figure 1.

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.

After stabilizing, a node notifies its successor of its identity. The notified member adopts the notifying member as its new predecessor if the notifying member is closer in identifier order than its current predecessor. Thus a stabilize operation always causes a notified operation. Figure 2 shows a sequence of join, stabilize, and notified operations in which a new member becomes part of the ring.

A machine ceases to become a member in a fail event, which also represents silently leaving the network. When a member fails, it no longer responds to queries from other members. Until a member fails, it is responsive to queries. Together these assumptions allow perfect failure detection. Also, it is assumed that successor lists are long enough so that a member is never left with no live successor in its list.

Failures can produce holes 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). 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 IV). 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 not in the ring, the ring must be reachable through best successors.

If any of these rules is violated, some members will not be reachable from some other members, and the ring-maintenance protocol cannot repair it.

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.

III. 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 II, 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. 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.

Lightweight analysis will require a limit on the number of nodes (potential members) analyzed, although this number does not affect the complexity of the model, i.e., the length of the program to be written, and need not be decided in advance. 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 program can be executed or analyzed (model-checked). During execution or analysis, a trace is generated by running all the process programs concurrently.
4) There are two forms of nondeterminism. Within a process program, the choice of guarded command is often nondeterministic. Also, during execution or analysis, the...

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.
interleaving of events in different processes is nondeterministic.

**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, and represents the time-varying state of an object 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. 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 form 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.

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 3 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.

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

**IV. 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 II, a necessary property is that a network be valid at all times, where the predicate valid is defined in Alloy as:

```alloy
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)
    } |
    { some ringMembers -- at most one ring
        all disj n1, n2, n3: ringMembers |
    }
}
```

By the time this paper is published, all models and analysis scripts will be available on the Web, so that others may benefit from detailed lessons learned.
valid \to \text{expected, as for those familiar with Alloy syntax. This pleasantness is to ordering of identifiers. It is concise and very readable, at least for those familiar with Alloy syntax. This pleasantness is to be expected, as \textit{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.}

\begin{verbatim}
assert JoinPreservesValidity {
  some Join && Valid[trace/first] => Valid[trace/last]
}
\end{verbatim}

In Promela a safety assertion can be inserted at any point in the code. For example, an assertion that the state is \textit{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, \textit{valid} is a C program that checks the model state for validity, and it is invoked with the following Promela statement:

\begin{verbatim}
assert c_expr{ valid(now.succ,now.succ2,now.prdc) }
\end{verbatim}

The \textit{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 III 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.\footnote{Promela inline macros work at the statement level and not the expression level. For this reason, \textit{between} on the left side of Figure 3 is actually C code.}

The correctness property for Chord is expressable in linear-time temporal logic, using the temporal progress operator \( \Diamond \) (\textit{eventually}). Mathematically, it 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.

Adding fairness to model-checking might solve this problem, but it would give a very distorted analysis of the protocol, which has no specified fairness or timing constraints. A better choice is to include a \textit{busyWait} variable that is initialized to false and set to true if any ineffective operation is ever executed. Then progress can be checked exclusively on traces where \textit{busyWait} is always false. Specifically, the correctness assertion for Spin is:

\begin{verbatim}
( <> [] churnStopped && ! busyWait ) => ( <> [] ideal )
\end{verbatim}

where \textit{churnStopped} is a Boolean variable that disables joins and failures, \( <> \) is \( \Diamond \), and \([]\) is the temporal operator \( \Box \) or \( \text{always} \).

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 V). For all practical purposes, progress properties cannot be asserted in Alloy.

\section{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 \textit{run} a \textit{predicate}. To search for counterexamples, you \textit{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 \textit{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 [9]. 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,\(^6\) but this was not the case.

Figure 4 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. First, the projection feature produces a time-sequence of snapshots, but the user interface shows only one at a time. Second, as with almost all graph-layout programs, the Analyzer displays each graph to optimize certain 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 4 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.\(^7\)

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 5 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 \textit{valid}, as the creators of Chord claimed to have proved it correct.

The process described in Figure 5 quickly found some easily fixed bugs, such as the one shown in Figure 4. 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 a post-state that is not \textit{valid} (a fatal error) and to work out manually a trace that reaches it from the initial state. This 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.

The report of this investigation\(^9\) discusses a “best” version of Chord with simple bugs fixed, optimistic assumptions about atomicity, and some improvements based on hints in the Chord papers. It was not possible to tell whether the “best” version is correct using Alloy, because the process of Figure 5 did not converge to either a useful global invariant or a

\(^6\)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.

\(^7\)There has been some work on using a model-checker to infer invariants—even using Chord as an example—but it was not very successful [10].
counterexample.

At this point it was necessary to start using Promela and Spin. Longer traces from Spin revealed that the “best” version is not correct (a typical example of these traces is the equivalent of about 50 events modeled in Alloy). 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.

VI. VERIFICATION

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 \textit{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
   \[ (\Diamond \Box \text{churnStopped} ) \implies (\Diamond \Box \text{Ideal} ) \]

We begin with showing that \textit{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 of ring symmetry.

My experimentation with Chord confirms this: small rings are pathological, larger rings “smooth out” and do not exhibit new problems. Note that the \textit{ring size} is defined as the number of members in the ring. The \textit{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.

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. 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. The experience recounted above indicates that node size 6 is the bare minimum for credible analysis of a version with a minimum ring size of 3.

Another dimension in which model-checking can be limited is the length of traces checked. For purposes of comparison, 1 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 big simplification, as analysis of the model with node size 5 found traces with lengths of 1.4 billion.

Another dimension in which 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 \textit{valid}, with node size 6, with trace length limit 1000, and in supertrace mode, yields the results in Table I. The \texttt{-w} option determines the size of the hash table. Although the run with \texttt{-w39} is not yet completed, from simple extrapolation it will take several more days, and the hash factor will not be close to 100.

Consequently, it seems that Spin cannot be used for a convincing proof of Chord correctness. Although understandable
in retrospect, it was initially a surprise that the presence of a global invariant gives Alloy such an important performance advantage.

The proof of the progress property for the Alloy model 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. 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, it cannot be used to verify the Chord progress property.

VII. SUMMARY OF COMPARISONS

It is important to stress that the comparisons reported here are valid only for the Chord project as described in Sections II through IV. The next section discusses how they might be extended to larger categories of project.

Table II summarizes the comparisons 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 also 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 no lightweight modeling at all, are far more extensive than this.

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 practice. As discussed in Section III, Alloy is as good for making a shared-memory model of a distributed system as Promela is. As discussed in Section V, it was necessary to hand-draw pictures from the outputs of both tools. As discussed in Section VI, automated verification of progress properties was not feasible in this project.

VIII. CONCLUSIONS

The claims of provable correctness made for the original version of Chord show 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 satisfactory for tricky network protocols.

This goal has not been achieved, even narrowing the meaning of “tricky network protocols” to “protocols very similar to Chord.” To find a correct version of Chord and prove it correct, it was necessary to use both Alloy and Spin, demanding a total amount of learning and real work that would be “daunting to a seasoned researcher, let alone a graduate student.”

On the other hand, using either Alloy or Spin, even for a short time, would be fun and cost-effective. It would have resulted in a far better specification of Chord and a far more realistic understanding of its properties than was available before this project.

Clearly the most urgent need identified by this work is the need for lightweight formal methods that can be used to complete an entire challenging engineering project. Researchers are already working as hard as they can on the performance of their tools. Some other capabilities that might have made the Chord problem easier are:

- Some optimizations or special features for analyzing traces in Alloy.
- Automated translation from graph properties in Alloy to C code.
- Accurate ways to estimate how big the analyses must be to produce sufficient confidence in the results.
- Best of all, some way to bridge the gap between generation of the reachable state space, with its strong advantages and disadvantages, and analysis with an invariant, with its complementary and equally strong characteristics.

Unfortunately, none of these capabilities is easy to provide.

The scope of this comparison is fairly broad with respect to the range of activities and goals involved in using lightweight formal methods. To get a complete picture of how well a formal method works in practice, it is necessary to consider modeling versus assertions, safety properties versus progress properties, and model exploration versus verification. The comparison does not include activities related to implementation, however, such as automated generation of test cases.

The scope of this comparison is narrow with respect to the systems being modeled. Even within the general area of distributed hash tables (DHTs) and other peer-to-peer networks, the scope could be extended in the following ways:

- Details of network communication could be included. Some Internet behaviors have a significant effect on Chord [12].
- Another strategy for making Chord correct (which was not used in this project) is to specify timing constraints that provide a modicum of fairness and thus prevent

<table>
<thead>
<tr>
<th>w</th>
<th>memory in megabytes</th>
<th>elapsed time</th>
<th>hash factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>256</td>
<td>3.14 hr</td>
<td>1.754</td>
</tr>
<tr>
<td>32</td>
<td>512</td>
<td>6.17 hr</td>
<td>1.774</td>
</tr>
<tr>
<td>33</td>
<td>1024</td>
<td>12.4 hr</td>
<td>1.787</td>
</tr>
<tr>
<td>39</td>
<td>65,536</td>
<td>3+ days</td>
<td>?</td>
</tr>
</tbody>
</table>

Table I: Spin safety checks in supertrace mode.
Promela + C / Spin  |  Alloy / Alloy Analyzer
--- | ---
**Real Advantages** | **Illusory Advantages**
not necessary to know a sufficiently strong global invariant (V) | the best choice for all concurrent distributed systems (III)
supports progress assertions (IV) | automated verification of progress assertions (VI)

**TABLE II**  
**SUMMARY OF COMPARATIVE ADVANTAGES.**

pathological scenarios. Timing constraints could be included.

- Two valuable properties of a DHT are *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. These properties do not seem to be required of the DHTs in common use, and may be very 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 be far easier to verify.

- The ring structure of Chord is a big factor in these results. Many other routing protocols operate on network graphs with fewer symmetries, and it would be interesting to study them.

It is not at all obvious what effect these extensions would have on the comparison.

At this point, it is difficult to lay out a research agenda for learning more about selection of formal methods. It seems that the best short-term approach would be more comparative studies such as this one, deep enough to really exercise the formal methods. Even a small number of additional studies might be enough so that the important factors begin to emerge from the haze of unimportant ones, and then subsequent work can be much better focused.

**ACKNOWLEDGMENTS**

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

**REFERENCES**