MIT Libraries | DSpace@MIT

MIT Open Access Articles

Alloy: a language and tool for exploring software designs

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Jackson, Daniel. "Alloy: a language and tool for exploring software designs." Communications of the ACM, 62, 9 (September 2019): 66-76 © 2019 The Author

As Published: 10.1145/3338843

Publisher: Association for Computing Machinery (ACM)

Persistent URL: https://hdl.handle.net/1721.1/129357

Version: Original manuscript: author's manuscript prior to formal peer review

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

Alloy: A Language and Tool for Exploring Software Designs

Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Alloy is a language and a toolkit for exploring the kinds of structures that arise in many software designs. This brief article aims to give a flavor of Alloy in action, to summarize how Alloy has been used to date, and thereby to give you a sense of how you might use it in your own software design work.

Formal Design Languages

Software involves structures of many sorts: architectures, database schemas, network topologies, ontologies, and so on. When you design a software system, you need to be able to express the structures that are essential to the design, and to check that they have the properties you expect.

You can express a structure by sketching it on a napkin. That's a good start, but it's limited. Informal representations give inconsistent interpretations, and they can't be analyzed mechanically. So people have turned to formal notations that define structure and behavior precisely and objectively, and that can exploit the power of computation.

By using formality early in development, you can minimize the costs of ambiguity and get feedback on your work by running analyses. The most popular approach that advocates this is agile development, in which the formal representation is code in a traditional programming language and the analysis is conventional unit testing.

As a language for exploring designs, however, code is imperfect. It's verbose and often indirect, and it doesn't allow *partial* descriptions in which some details are left to be resolved later. And testing, as a way to analyze designs, leaves much to be desired. It's notoriously incomplete, and burdensome, since you need to write test cases explicitly. And it's very hard to use code to articulate design without getting mired in low level details (such as the choice of data representations).

An alternative, which has been explored since the 1970s, is to use a design language built not on conventional machine instructions but on logic. Partiality comes for free because, rather than listing each step of a computation, you write a logical constraint saying what's true after, and that constraint can say as little or as much as you please. To analyze such a language, you use specialized algorithms such as model checkers or satisfiability solvers (more on these below). This usually requires much less effort than testing, since you only need to express the property you want to check rather than a large collection of cases. And the analysis is much more complete than testing, because it effectively covers all (or almost all) test cases that you could have written by hand.

What Came Before: Theorem Provers and Model Checkers

To understand Alloy, it helps to know a bit about the context in which it was developed, and the tools that existed at the time.

Theorem provers are mechanical aids for constructing mathematical proofs. To apply a theorem prover to a software design problem, you formulate some intended property of the design, and then attempt to prove the theorem that the property follows from the design. Theorem provers tend to provide very rich logics, so they can usually express any property you might care about, at least about states and state transitions—more dynamic properties can require a temporal logic that theorem provers don't typically support directly. Also, because they generate mathematical proofs, which can be checked by tools that are smaller and simpler than the tool that finds the proof, you can be confident that the analysis is sound.

On the other hand, the combination of an expressive logic and sound proof has meant that finding proofs cannot generally be automated. So theorem provers usually require considerable effort and expertise from the user, often orders of magnitude greater than the effort of constructing a formal design in the first place. Moreover, failure to find a proof does not mean that a proof does not exist, and theorem provers don't provide counterexamples that explain concretely why a theorem is not valid. So theorem provers are not so useful when the intended property does not hold—which unfortunately is the common case in design work.

Model checkers revolutionized design analysis by providing exactly the features theorem provers lacked. They offer push-button automation, requiring the user to give only the design and property to be checked. They allow dynamic properties to be expressed (through temporal logics), and generate counterexamples when properties do not hold. Model checkers work by exploring the space of possible states of a system, and if that space is large, they may require considerable computational resources (or may fail to terminate). The so-called "state explosion" problem arises because model checkers are often used to analyze designs involving components that run in parallel, resulting in an overall state space that grows exponentially with the number of components.

Alloy was inspired by the successes and limitations of model checkers. For designs involving parallelism and simple state (comprising boolean variables, bound-

To appear, CACM
Draft of February 22, 2019

ed integers, enumerations and fixed-size arrays), model checkers were ideal. They could easily find subtle synchronization bugs that appeared only in rare scenarios that involved long traces with multiple context switches, and therefore eluded testing.

For hardware designs, model checkers were often a good match. But for software designs they were less ideal. Although some software design problems involve this kind of synchronization, often the complexity arises from the structure of the state itself. Early model checkers (such as SMV [9]) had limited expressiveness in this regard, and did not support rich structures such as trees, lists, tables and graphs.

Explicit state model checkers, such as SPIN [14], and later Java Pathfinder [37], allowed designs with rich state to be modeled, but, despite providing support for temporal properties, gave little help for expressing structural ones. To express reachability (for example that two social media users are connected by some path of friend edges), you would typically need to code an explicit search, which would have to be executed at every point at which the property was needed. Also, explicit state model checkers have limited support for partiality (since the model checker would have to conduct a costly search through possible next states to find one satisfying the constraints).

Particularly hard for all model checkers are the kinds of designs that involve a *configuration* of elements in a graph or tree structure. Many network protocols are designed to work irrespective of the initial configuration (or of the configuration as it evolves), and exposing a flaw often involves not only finding a behavior that breaks a property but also finding a configuration in which to execute it.

Even the few model checkers that can express rich structures are generally not up to this task. Enumerating possible configurations is not feasible, because the number of configurations grows super-exponentially: if there are n nodes, there are $2^{n \times n}$ ways to connect them.

Alloy's Innovations

Alloy brought a new kind of design language and analysis, made possible by three innovations.

Relational logic. Alloy uses the same logic for describing designs and properties. This logic combines the for-all and exists-some quantifiers of first-order logic with the operators of set theory and relational calculus.

The idea of modeling software designs with sets and relations had been pioneered in the Z language [32]. Alloy incorporated much of the power of Z, while simplifying the logic to make it more tractable.

First, Alloy allows only first-order structures, ruling out sets of sets and relations over sets, for example. This changes how designs are modeled, but not what can be modeled; after all, relational databases have flourished despite being first order.

Second, taking advantage of this restriction, Alloy's operators are defined in a very general way, so that most expressions can be written with just a few operators. The key operator is relational join, which in conventional mathematics only applies to binary relations, but in Alloy works on relations of any arity. By using a dot to represent the join operator, Alloy lets you write dereferencing expressions as you would in an object oriented programming language, but gives these expressions a simple mathematical interpretation. So, as in Java, given an employee *e*, a relation *dept* that maps employees to departments, and a relation *manager* that maps departments to their managers, *e.dept.manager* would give the manager of *e*'s department. But unlike in Java, the expression will also work if *e* is a set of employees, or *dept* can map an employee to multiple departments, giving the expected result—the set of managers of the set of departments that the employees *e* belong to. The expression *dept.manager* is well defined too, and means the relation that maps employees to their managers. You can also navigate backwards, writing *manager.m* for the department(s) that *m* manages.

(A note for readers interested in language design: this flexibility is achieved by treating all values as relations—a set being a relation with one column, and a scalar being a set with one element—and defining a join operator that applies uniformly over a pair of relations, irrespective of their arity. In contrast, other languages tend to have multiple operators, implicit coercions or overloading to accommodate variants that Alloy unifies.)

Alloy was influenced also by modeling languages such as UML. Like the class diagrams of UML, Alloy makes it easy to describe a universe of objects as a classification tree, with each relation defined over nodes in this tree. Alloy's dot operator was inspired in part by the navigational expressions of OCL (the Object Constraint Language [39] of UML), but by defining the dot as relational join, Alloy dramatically simplifies the semantics of navigation.

Small scope analysis. Even plain first-order logic (without relational operators) is not decidable. This means that no algorithm can exist that could analyze a software design written in a language like Alloy completely. So something has to give. You could make the language decidable, but that would cripple its expressive power and make it unable to express even the most basic properties of structures (although exciting progress has been made recently in applying decidable fragments of first-order logic to certain problems [29]). You could give up on automation, and require help from the user, but this eliminates most of the benefit of an analysis tool; analysis is no longer a reward for constructing a design model, but a major extra investment beyond modeling.

The other option is to somehow limit the analysis. Prior to Alloy, two approaches were popular. *Abstraction* reduces the analysis to a finite number of cases, by introducing abstract values that each correspond to an entire set of real values. This often results in false positives that are hard to interpret, and in practice picking the right abstraction calls for considerable ingenuity. *Simulation* picks a finite number of cases, usually by random sampling, but it covers such a small part of the state space that subtle flaws elude detection.

Alloy offered a new approach: running all small tests. The designer specifies a *scope* that bounds each of the types in the specification. A scope of 5, for example, would include tests involving at most 5 elements of each type: 5 network nodes, 5 packets, 5 identifiers, and so on.

The rationale for this is the *small scope hypothesis*, which asserts that most bugs can be demonstrated with small counterexamples. That means that if you test for all small counterexamples, you are likely

```
1  abstract sig EndPoint { }
2  sig Server extends EndPoint {
3    causes: set HTTPEvent
4    }
5  sig Client extends EndPoint { }
6  abstract sig HTTPEvent {
7    from, to, origin: EndPoint
8    }
9  sig Request extends HTTPEvent {
10   response: lone Response
11   }
12 sig Response extends HTTPEvent {
13   embeds: set Request
14   }
15 sig Redirect extends Response {
16   }
```

FIG. 1 Structure declarations

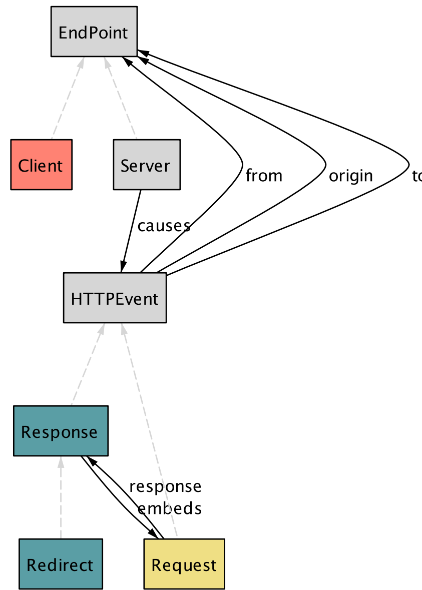

FIG. 2 Data model from declarations

```
17 fact Directions {
18   Request.from + Response.to in Client
19   Request.to + Response.from in Server
20   }
21 fact RequestResponse {
22   all r: Response | one response.r
23   all r: Response |
24     r.to = response.r.from
25     and r.from = response.r.to
26   all r: Request |
27     r not in r.^(response.embeds)
28   }
29 fact Causality {
30   all e: HTTPEvent, s: Server |
31     e in s.causes iff e.from = s or
32       some r: Response |
33         e in r.embeds and r in s.causes
34   }
35 fact Origin {
36   all r: Response, e: r.embeds |
37     e.origin = r.origin
38   all r: Response | r.origin =
39     (r in Redirect implies
40       response.r.origin else r.from)
41   all r: Request |
42     no embeds.r implies
43       r.origin in r.from
44   }
45 pred EnforceOrigins (s: Server) {
46   all r: Request |
47     r.to = s implies
48       r.origin = r.to or r.origin = r.from
49   }
```

FIG. 3 Fact and predicate declarations

to find any bug. Many Alloy case studies have confirmed the hypothesis, by performing an analysis in a variety of scopes and showing, retrospectively, that a small scope would have sufficed to find all the bugs that were discovered.

Translation to SAT. Even with small scopes, the state space of an Alloy model is fiendishly large. The state comprises a collection of variables whose values are relations. Just one binary relation in a scope of 5 has $5 \times 5 = 25$ possible edges, and thus 2^{25} possible values. A very small design might have 5 such relations, giving $(2^{25})^5$ possible states—about 10^{37} states. Even checking a billion cases per second, such an analysis would take many times the age of the universe.

Alloy therefore does not perform an explicit search, but instead translates the design problem to a *satisfiability problem* whose variables are not relations but simple bits. By flipping bits individually, a satisfiability (SAT) solver can usually find a solution (if there is one) or show that none exists by examining only a tiny portion of the space.

Alloy's analysis tool is essentially a compiler to SAT, which allows it to exploit the latest advances in SAT solvers. The success of SAT solvers has been a remarkable story in computer science: theoreticians had shown that SAT was inherently intractable, but it turned out that most of the cases that arise in practice can be solved efficiently. So SAT went from being the archetypal insoluble problem used to demonstrate the infeasibility of other problems to being a soluble problem that other problems could be translated to. Alloy also applies a variety of tactics to reduce the problem prior to solving, most notably adding *symmetry breaking constraints* that save the SAT solver from considering cases that are equivalent to one another.

Example: Modeling Origins

To see Alloy in action, let's explore the design of an origin-tracking mechanism for web browsers. The model shown here is a toy version of a real model that exposed several serious flaws in browser security [1]. Although it cuts corners and is unrealistic in some respects, it does capture the spirit and style of the original model, and is fairly representative of how Alloy is often used.

First, some background for those unfamiliar with browser security. Cross-site request forgery (CSRF) is a pernicious and subtle attack in which a malicious script running in a page that the user has loaded makes a hidden and unwanted request to a website for which the user is already authenticated. This may happen either because the user was enticed to load a page from a malicious server, or because a supposedly safe server was the subject of a cross-site scripting attack, and served a page containing a malicious script. Such a script can issue any request the user can issue; one of the first CSRF vulnerabilities to be discovered, for example, allowed an attacker to change the delivery address for the user's account in a DVD rental site. What makes CSRF particularly problematic is that the browser sends authentication credentials stored as cookies spontaneously when a request is issued, whether that request is made explicitly by the user or programmatically by a script.

One way to counter CSRF is to track the origins of all responses received from servers. In our example, the browser would mark the malicious script as originating at the malicious or compromised server. The subsequent request made by that script to the rental site server—the target of the attack—would be labeled as having this other origin. The target server can be configured so that it only accepts requests that originate directly from the user (for example, by the user entering the URL for the request in the address bar), or from itself (for example, from a

```
50  check {
51  no good, bad: Server {
52    good.EnforceOrigins
53    no r: Request |
54      r.to = bad and r.origin in Client
55    some r: Request |
56      r.to = good and r in bad.causes
57    }
58  } for 5
```

FIG. 4 Check command

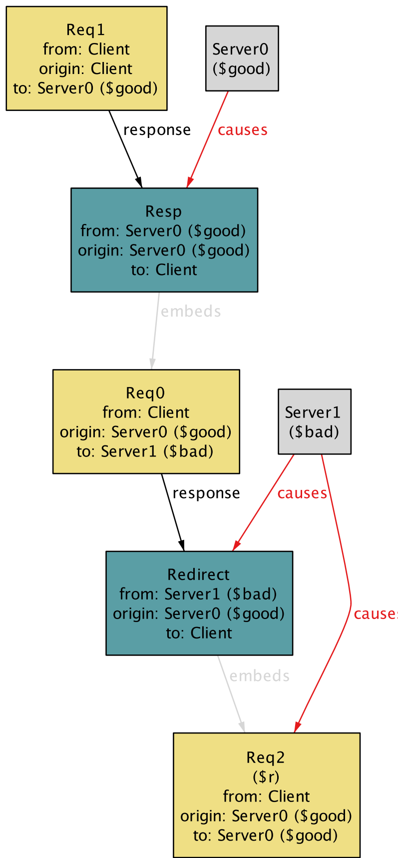

FIG. 5 Counterexample for check of Fig. 4

script embedded in a page previously sent by the target server). As always the devil is in the details, and we shall see that a plausible design of this mechanism turns out to be flawed.

Here are some features to look out for in this model, which distinguish Alloy from many other approaches:

- A rich structure of objects, classification and relationships;
- Constraints in a simple logic that exploits the relations and sets of the structure, avoiding the kind of low level structures (arrays and indices, etc.) that are often needed in model checkers;
- Capturing dynamic behavior without any need for a built-in notion of time or state;
- Intended properties to check expressed in the same language as the model itself;
- An abstract style of modeling that includes only those aspects essential to the problem at hand.

We start by declaring a collection of *signatures* (Fig. 1). A signature introduces a set of objects and some fields that relate them to other objects. So *Server*, for example, will represent the set of server nodes, and has a field *causes* that associates each server with the set of HTTP events that it causes.

Keywords (or their omission) indicate the multiplicity of the relations between objects: thus each HTTP event has exactly one *from* endpoint, one *to* endpoint, and one *origin* endpoint (line 7); each request has at most one response (line 10, with *lone* being read as "less than or equal to one"); and each response embeds any number of requests (line 13).

Objects are, mathematically, just atomic identifiers without any internal structure. So the *causes* relation includes tuples of the form *(s, e)* where the value of *s* is some atomic identifier representing a server object, and the value of *e* is some atomic identifier representing an event.

Fields are declared in signatures to allow a kind of object-oriented mindset. Alloy supports this by resolving field names contextually (so that field names need not be globally unique), and by allowing "signature facts" (not used here) that are implicitly scoped over the elements of a signature and their fields. But don't be misled into thinking that there is some kind of complex object semantics here. The signature structure is only a convenience, and just introduces a set and some relations.

The *extends* keyword defines one signature as a subset of another. An *abstract* signature has no elements that do not belong to a child signature, and the extensions of a signature are disjoint. So the declarations of *EndPoint*, *Server* and *Client* imply that the set of endpoints is partitioned into servers and clients: no server is also a client, and there is no endpoint that is neither client nor server. A relation defined over a set applies over its subsets too, so the declaration of *from*, for example, which says that every HTTP event is from a single endpoint, implies that the same is true for every request and response. (Alloy is best viewed as untyped. It turns out that conventional programming language types are far too restrictive for a modeling language. Alloy thus allows expressions such as *HTTPEvent.response*, denoting the set of responses to any events, but its type checker rejects an expression such as *Request.embeds* which always denotes an empty set [12].)

The Alloy Analyzer can generate a graphical representation of the sets and relations from the signature declarations (Fig. 2); this is just an alternative view and involves no analysis.

Moving to the substance of what the model actually means:

- The *from* and *to* fields are just the source and destination of the event's packet.
- For a response *r*, the expression *r.embeds* denotes a set of requests that are embedded as JavaScript in the response; when that response is loaded into the browser, the requests are executed spontaneously.
- A *redirect* is a special kind of response that indicates that a resource has moved, and spontaneously issues a request to a different server; this second request is modeled as an embedded request in the redirect response.
- The *origin* of an event is a notion computed by the browser as a means of preventing cross-site attacks. As we'll see later, the idea is that a server may choose to reject an event unless it originated at that server or at a browser.
- The *cause* of an event is not part of the actual state of the mechanism. It is introduced in order to express the essential design property: that an evil server cannot cause a client to send a request to a good server.

Now let's look at the constraints (Fig. 3). If there were no constraints, any behavior would be possible; adding constraints restricts the behavior to include only those that are intended by design.

The constraints are grouped into separate named *facts* to make the model more understandable:

- The *Directions* fact contains two constraints. The first says that every request is from, and every response is to, a client; the second says that every request is to, and every response is from

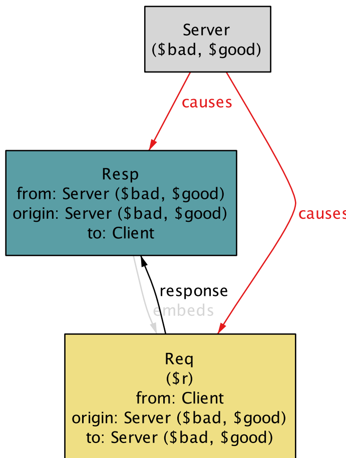

FIG. 6 A bogus counterexample

a server. These kinds of constraints can be written in many ways. Here we've chosen to use expressions denoting sets of endpoints—*Request.from* for the set of endpoints that requests are from, eg. But we could equally well have written a constraint like

```
from in
  Request -> Client + Response -> Server
```

to say that the *from* relation maps requests to clients and responses to servers. Or in a more familiar but less succinct style, we could have used quantifiers:

```
all r: Request | r.from in Client
all r: Response | r.from in Server
```

(which constrains only the range of the relations, which is sufficient in this case since the declarations constrain their domains).

- The *RequestResponse* fact defines some basic properties of how requests and responses work: that every response is from exactly one server (line 22); that every response is to the endpoint its request was from, and from the endpoint its request was to (line 23); and that a request cannot be embedded in a response to itself (line 26). Two expressions in these constraints merit explanation. The expression *response.r* exploits the flexibility of the join operator to navigate backwards from the response *r* to the request it responds to; it could equivalently be written *r.~response* using the transpose operator ~. The expression *r.^(response.embeds)* starts with the request *r*, and then applies to it one or more navigations (using the closure operator ^) of following the *response* and *embeds* relations, as if we'd written instead the infinite expression

```
r.response.embeds
+ r.response.embeds.response.embeds
+ r.response.embeds.response.embeds
  .response.embeds
+ …
```

defining the requests embedded in the response to *r*, the requests embedded in the response to the requests embedded in the response to *r*, and so on. (Equivalently, *r.^p* is the set of nodes reachable from *r* in the graph whose edges correspond to the relation *p*.)

- The *Causality* fact defines the *causes* relation. It says that an event is caused by a server if and only if it is from that server, or is embedded in a response that the server causes.
- The *Origin* fact describes the origin-tracking mechanism. Each constraint defines the origin of a different kind of HTTP event. The first (line 36) says that every embedded request *e* has the same origin as the response *r* that it is embedded in. The second (line 38) defines the origin of a response: it says that if the response is a redirect, it has the same origin as the original request, and otherwise its origin is the server that the response came from. The third (line 41) handles a request that is not embedded: its origin is the endpoint it comes from (which will usually be the browser).

Finally, *EnforceOrigins* is a *predicate* that can be applied to a server, indicating that it chooses to enforce the origin header, allowing incoming requests only if they originate at that server, or at the client that sent the request.

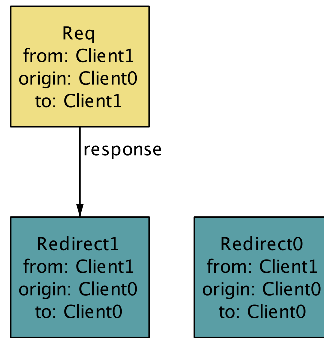

FIG. 7 A simulated instance

With all this in place—the structure of endpoints and messages, the rules about how origins are computed and used, and the definition of causality—we can define a design property to check (Fig. 4).

The keyword *check* introduces a *command* that can be executed. This command instructs the Alloy Analyzer to search for a refutation for the given constraint. In this case, the constraint asserts the non-existence of a cross-site request forgery attack; refuting this will show that the origin mechanism is not designed correctly, and an attack is possible.

The constraint says that there are no two servers, *good* and *bad*, such that the good server enforces the origin header (line 52), there are no requests sent directly to the bad server that originate in the client (line 53), and yet there is some request to the good server that was caused by the bad server (line 55).

Analysis Results: Finding Bugs

The Alloy Analyzer finds a counterexample (Fig. 5) almost instantaneously—in 30ms on my 2012 Mac Book (with a 2.6 GHz i7 processor and 16GB of RAM).

The counterexample can be displayed in various ways—as text, as a table, or as a graph whose appearance can be customized. I've chosen the graph option, and have selected which objects are to appear as nodes (just the events and the servers), which relations are to appear as edges (those between events, and *causes*), and I've picked colors for the sets and relations. I've also chosen to use the Skolem constants (witnesses that the analyzer finds for the quantified variables) *good* and *bad* to label the servers.

Reading the graph from the top, looking just at the large rectangles representing the HTTP events, we see that a request (*Req1*) was sent from a client to the good server. The response (*Resp*) embeds a request (*Req0*) that is sent to the bad server; this is a cross-site request which won't be rejected because the bad server accepts incoming requests irrespective of origin. The bad server's response is to send a redirect whose embedded request (*Req2*) is received by the good server. (Note that the numbering of objects is arbitrary: *Req1* actually happens before *Req0*.)

Now looking at the server nodes and the events they cause, we see that, as ex-

pected, the good server caused the response to the first request, and the bad server caused the redirect and its subsequent embedded request. The problem is the mismatch between cause and origin in the last request (*Req2*): we can see that it was *caused* by the bad server, but it was labelled as originating at the good server. In other words, the origin tracking design is allowing a cross-site request forgery by incorrectly identifying the origin of the request in the redirect.

The solution to this problem turns out to be non-trivial. Updating the origin header after each redirect would fail for websites that offer open redirection; a better solution is to list a chain of endpoints in the origin header [1].

Agile Modeling

As I mentioned earlier, our model is representative of many Alloy models. But the way I presented it was potentially misleading. In practice, users of Alloy don't construct a model in its entirety and then check its properties. Instead, they proceed in a more agile way, growing the model and simulating and checking it as they go.

Take, for example, the constraint on line 26 of Fig. 3. Initially, I hadn't actually noticed the need for this constraint. But when I ran the check for the first time (without this constraint), the analyzer presented me with counterexamples such as the one shown in Fig. 6, in which the response to a request is the very response in which the request is embedded!

One way to build a model, exploiting Alloy's ability to express and analyze very partial models, is to add one constraint at a time, exploring its effect. You don't need to have a property to check; you can just ask for an instance of the model satisfying all the constraints.

Doing this even before any explicit constraints have been included is very helpful. You can run just the data model by itself and see a series of instances that satisfy the constraints implicit in the declarations. Often doing this alone exposes some interesting issues. In this case, the first few instances include examples with no HTTP events, and with requests and responses that are disconnected.

To get more representative instances, you can specify an additional constraint to be satisfied. For example, the command

```
run {some response}
```

will show instances in which the *response* relation has some tuples. The first one generated (Fig. 7) shows a request with a response that is a redirect from the same source as the request, and sent to an endpoint that is also its origin, and it includes an orphaned redirect unrelated to any request! These anomalies immediately suggest enrichments of the model.

When we developed Alloy, we underestimated the value of this kind of simulation. As we experimented with Alloy, however, we came to realize how helpful it is to have a tool that can generate provocative examples. These examples invariably expose basic misunderstandings, not only about what's being modeled but also about which properties matter. It's essential that Alloy provides this simulation for free: in particular, you don't need to formulate anything like a test case, which would defeat the whole point.

Growing a model in a declarative language like Alloy is very different from growing a program in a conventional programming language. A program starts with no behaviors at all, and as you add code, new behaviors become possible. With Alloy, it's the opposite. The empty model, since it lacks any constraints, allows every possible behavior; as you add constraints, behaviors are eliminated.

This allows a powerful style of incremental development in which you only add constraints that are absolutely essential for the task at hand—whether that's eliminating pathological cases or ensuring that a design property holds.

Typically a model includes both a description of the mechanism being designed and some assumptions about the environment in which it operates. Our model does not separate these rigorously, but where brevity is not such a pressing concern, it would be wise to do so. We could separate, for example, the constraints that model the setting and checking of the origin field from those that describe what kinds of requests and responses are possible.

Obviously, the less you assume about the environment, the better, since every assumption you make is a risk (since it may turn out to be untrue). In our model, for example, we don't require every request to have a response. It would be easy to do—just change the declaration of *response* in line 10 of Fig. 1 by dropping the *lone* keyword—but would only make the result of the analysis less general. Likewise, the less you constrain the mechanism, the better. Allowing multiple behaviors gives implementation freedom, which is especially important in a distributed setting.

Simulation matters for a more profound reason. Verification—that is, checking properties—is often overrated in its ability to prevent failure. As Christopher Alexander explains [2], designed artifacts usually fail to meet their purposes not because specifications are violated but because specifications are unknown. The "unknown unknowns" of a software design are invariably discovered when the design is finally deployed, but can often be exposed earlier by simulation, especially in the hands of an imaginative designer.

Verification, in contrast, is too narrowly focused to produce such discoveries. This is not to say that property checking is not useful—it's especially valuable when a property can be assured with high confidence using a tool such as Alloy or a model checker or theorem prover (rather than by testing). But its value is always contingent on the sufficiency of the property itself, and techniques that help you explore properties have an important role to play.

Uses of Alloy

Hundreds of papers have reported on applications of Alloy in a wide variety of settings. Here are some examples to give a flavor of how Alloy has been used.

Critical systems. A team at the University of Washington constructed a dependability case [18] for a neutron radiotherapy installation. They devised an ingenious technique for verifying properties of code against specifications using lightweight, pluggable checkers. The end-to-end dependability case was assembled in Alloy from the code specifications, properties of the equipment and environment, and the expected properties, and then checked using the Alloy Analyzer. The analysis found several safety-critical flaws in the latest version of the control software, which the researchers were able to correct prior to its deployment. For a full description, see a recent research report

[30] and additional information on the project's website [36].

Network protocols. Pamela Zave, a researcher at AT&T, has been using Alloy for many years to construct and analyze models of networking, and for designing a new unifying network architecture. In a major case study, she analyzed Chord, a distributed hash table for peer-to-peer applications. The original paper on Chord [33]—one of the most widely cited papers in computer science—notes that an innovation of Chord was its relative simplicity, and consequently the confidence users can have in its correctness. By modeling and analyzing the protocol in Alloy, Zave showed that the Chord protocol was not, however, correct, and she was able to develop a fixed version that maintains its simplicity and elegance while guaranteeing correct behavior [43]. Zave also used the explicit model checker SPIN [14] in this work, and wrote an insightful article explaining the relative merits of the two tools, and how she used them in tandem [42].

Web security. The demonstration example of this paper is drawn from a real study performed by a research group at Berkeley and Stanford [1]. They constructed a library of Alloy models to capture various aspects of web security mechanisms, and then analyzed five different mechanisms, including: WebAuth, a web-based authentication protocol based on Kerberos deployed at several universities including Stanford; HTML5 forms; the Cross-Origin Resource Sharing protocol; and proposed designs for using the referer header and the origin header to foil cross-site attacks (of which the last is the basis for the example here). The base library was written in 2,000 lines of Alloy; the various mechanisms required between 20 and 214 extra lines; and every bug was found within two minutes and a scope of 8. Two previously known vulnerabilities were confirmed by the analysis, and three new ones discovered.

Memory models. John Wickerson and his colleagues have shown that four common tasks in the design of memory models—generating conformance tests, comparing two memory models, checking compiler optimizations, and checking compiler mappings—can all be framed as constraint satisfaction problems in Alloy [41]. They were able to reproduce automatically several results for C11 (the memory model introduced in 2011 for C and C++) and common compiler optimizations associated with it, for the memory models of the IBM Power and Intel x86 chips, and for compiler mappings from OpenCL to AMD-style GPUs. They then used their technique to develop and check a new memory model for Nvidia GPUs.

Code verification. Alloy can also be used to verify code, by translating the body of a function into Alloy, and asking Alloy to find a behavior of the function that violates its specification. Greg Dennis built a tool called Forge that wraps Alloy so that it can be applied directly to Java code annotated with JML specifications. In a case study application [10], he checked a variety of implementations of the Java collections list interface, and found bugs in one (a GNU Trove implementation). Dennis also applied his tool to KOA, an electronic voting system used in the Netherlands that was annotated with JML specifications and had previously been analyzed with a theorem proving tool, and found several functions that did not satisfy their specifications [11].

Civil engineering. In one of the more innovative applications of Alloy, John Baugh and his colleagues have been applying Alloy to problems in large-scale physical simulation. They designed an extension to ADCIRC—an ocean circulation model widely used by the U.S. Army Corps of Engineers and others for simulating hurricane storm surge—that introduces a notion of subdomains to allow more localized computation of changes (and thus reduced overall computational effort). Their extension, which has been incorporated into the official ADCIRC release, was modeled and verified in Alloy [7].

Alloy as a backend. Because Alloy offers a small and expressive logic, along with a powerful analyzer, it has been exploited as a backend in many different tools. Developers have often used Alloy's own engine, Kodkod [34], directly, rather than the API of Alloy itself, because it offers a simpler programmatic interface with the ability to set bounds on relations, improving performance. Jasmine Blanchette's Nitpick tool [8], for example, uses Kodkod to find counterexamples in Isabelle/HOL, saving the user the trouble of trying to prove a theorem that is not true, and the Margrave tool [26] analyzes firewall configurations. Last year, a team from Princeton and Nvidia built a tool that uses Alloy to synthesize security attacks that exploit the Spectre and Meltdown vulnerabilities [35].

Teaching. Alloy has been widely taught in undergraduate and graduate courses for many years. At the University of Minho in Portugal, Alcino Cunha teaches an annual course on formal methods using Alloy, and has developed a web interface to present students with Alloy exercises (which are then automatically checked). At Brown University, Tim Nelson teaches *Logic for Systems*, which uses Alloy for modeling and analysis of system designs, and has become one of the most popular undergraduate classes. Because the Alloy language is very close to a pure relational logic, it has also been popular in the teaching of discrete mathematics, for example in a course that Charles Wallace teaches at Michigan Technological University [38] and appearing as a chapter in a popular textbook [15].

Alloy Extensions

Many extensions to Alloy—both to the language and to the tool—have been created. These offer a variety of improvements in expressiveness, performance and usability. For the most part, these extensions have been mutually incompatible, but a new open source effort is now working to consolidate them. There are too many efforts to include here, so we focus on representatives of the main classes.

Higher-order solving. The Alloy Analyzer's constraint solving mechanism cannot handle formulas with universal quantifications over relations—that is, problems that reduce to "find some relation P such that for every relation Q…" This is exactly the form that many synthesis problems take, in which the relation P represents a structure to be synthesized, such as the abstract syntax tree of a program, and the relation Q represents the state space over which certain behaviors are to be verified. Alloy* [24] is an extension of Alloy that can solve such formulas, by generalizing a tactic known as counterexample-guided inductive synthesis that has been widely used in synthesis engines.

Temporal logic. Alloy has no built-in notion of time or dynamic behavior. On

the one hand, this is an asset, because it keeps the language simple, and allows it to be used very flexibly. We exploited this in the example model of this paper, where the flow of time is captured in the *response* relation that maps each request to its response. By adding a signature for state, Alloy supports the specification style common in languages such as B, VDM and Z; and by adding a signature for events, Alloy allows analysis over traces that can be visualized as series of snapshots. On the other hand, it would often be preferable to have dynamic features built into the language. Electrum [20] extends Alloy with a keyword *var* to indicate that a signature or field has a time-varying value, and with the quantifiers of linear temporal logic (which fit elegantly with Alloy's traditional quantifiers). DynAlloy [31] offers similar functionality, but using dynamic logic instead, and is the basis of an impressive code analysis tool called TACO [13] that outperforms Forge (mentioned above) by employing domain-specific optimizations. No extension of Alloy, however, has yet addressed the problem of combining Alloy's capacity for structural analysis with the ability of traditional model checkers to explore long traces, so Alloy analyses are still typically limited to short traces.

Instance generation. The result of an Alloy analysis is not one but an entire set of solutions to a constraint-solving problem, each of which represents either a positive example of a scenario, or a negative example, showing how the design fails to meet some property. The order in which these appear is somewhat arbitrary, being determined both by how the problem is encoded and the tactics of the backend SAT solver. Since SAT solvers tend to try false before true values, the instances generated tend to be small ones—with few nodes and edges. This is often desirable, but is not always ideal. Various extensions to the Alloy Analyzer provide more control over the order in which instances appear. Aluminum [28] presents only minimal scenarios, in which every relation tuple is needed to satisfy the constraints, and lets the user add new tuples, automatically compensating with a (minimal) set of additional tuples required for consistency. Amalgam [27] lets users ask about the provenance of an instance, indicating which subformula is responsible for requiring (or forbidding) a particular tuple in the instance. Another extension [21] of the Alloy Analyzer generates minimal and maximal instances, and choosing a next instance that is as close to, or as far away from, the current instance as possible.

Better numerics. Alloy handles numerical operations by treating numbers as bit strings. This has the advantage of fitting into the SAT solving paradigm smoothly, and it allows a good repertoire of integer operations. But the analysis scales poorly, making Alloy unsuitable for heavily numeric applications. The finite scopes of Alloy can also be an issue when a designer would like numbers to be unbounded. A possible solution is to replace the SAT backend with an SMT backend instead. This is challenging because SMT solvers have not traditionally supported relational operators. Nevertheless, a team at the University of Iowa has recently extended CVC4, a leading SMT solver, with a theory of finite relations, and has promisingly demonstrated its application to some Alloy problems [23].

Configurations. Many Alloy models contain two loosely coupled parts, one defining a configuration (say of a network) and the other the behavior (say of sending packets). By iterating through configurations and analyzing each independently, one can often dramatically reduce analysis time [22]. In some applications, a configuration is already fully or partially known, and the goal is to complete the instance—in which case searching for the configuration is a wasted effort. Kodkod, Alloy's engine, allows the explicit definition of a "partial instance" to support this, but in Alloy itself, this notion is not well supported (and relies on a heuristic for extracting partial instances from formulas in a certain form). Researchers have therefore proposed a language extension [25] to allow partial instances to be defined directly in Alloy itself.

How to Try Alloy

The Alloy Analyzer [3] is a free download available for Mac, Windows and Linux. The Alloy book [16] provides a gentle introduction to relational logic and to the Alloy language, gives many examples of Alloy models, and includes a reference manual and a comparison to other languages (both of which are available on the book's website [17]). The Alloy community answers questions tagged with the keyword *alloy* on StackOverflow, and hosts a discussion forum [5]. A variety of tutorials for learning Alloy are available online too, as well as blog posts with illustrative case studies and examples (eg [40, 19]). The model used in this paper is available (along with its visualization theme) in the Alloy community's model repository [4].

Acknowledgments

I am very grateful to David Chemouil, Alcino Cunha, Peter Kriens, Shriram Krishnamurthi, Emina Torlak, Hillel Wayne, Pamela Zave, and the anonymous reviewers, whose suggestions improved this paper greatly; to Moshe Vardi, who encouraged me to write it; and to Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell and Dawn Song whose work formed the basis of the paper's example. Thank you also to the many members of the Alloy community who have contributed to Alloy over the years.

References

1. Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell and Dawn Song. Towards a Formal Foundation of Web Security. *23rd IEEE Computer Security Foundations Symposium*, Edinburgh, 2010, pp. 290–304.
2. Christopher Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1964.
3. *Alloy Tools website*: http://alloytools.org.
4. *Alloy Models repository*: https://github.com/AlloyTools/models
5. *Alloy discussion forum*: https://groups.google.com/forum/#!forum/alloytools
6. Adam Barth, Colin Jackson and John C. Mitchell. Robust defenses for cross-site request forgery. *15th ACM Conf. on Computer and Communications Security* (CCS 2008). ACM, 2008, pp. 75–88.
7. John Baugh and Alper Altuntas. Formal methods and finite element analysis of hurricane storm surge: A case study in software verification. *Science of Computer Programming*, 158:100–121, 2018.
8. Jasmine Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *First International Conference on Interactive Theorem Proving* (ITP 2010), M. Kaufmann and L.C. Paulson, eds. LNCS 6172, pp. 131–146, Springer, 2010.
9. Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Fifth Annual Symposium on Logic in Computer Science* (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990, pp. 428–439.
10. Greg Dennis, Felix Chang and Daniel Jackson. Modular Verification of Code with SAT. *Inter-*

national Symposium on Software Testing and Analysis. Portland, ME, July 2006.

11. Greg Dennis, Kuat Yessenov and Daniel Jackson. Bounded Verification of Voting Software. *Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments* (VSTTE 2008) . Toronto, Canada, October 2008.
12. Jonathan Edwards, Daniel Jackson and Emina Torlak. A type system for object models. *12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004, pages 189–199, 2004.
13. Juan P. Galeotti, Nicolas Rosner, Carlos G. Lopez Pombo and Marcelo F. Frias. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *IEEE Trans. Softw. Eng.* 39, 9 (September 2013), pp. 1283–1307.
14. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*, Addison Wesley, 2003.
15. Michael Huth and Mark Ryan. *Logic in Computer Science: Modeling and Reasoning about Systems*, Cambridge University Press, 2004.
16. Daniel Jackson. *Software Abstractions*, MIT Press, Second edition, 2012.
17. Daniel Jackson. *Software Abstractions* website. http://softwareabstractions.org.
18. Daniel Jackson, Martyn Thomas, and Lynette I. Millett, eds. *Software For Dependable Systems: Sufficient Evidence?* Committee on Certifiably Dependable Software Systems, Computer Science and Telecommunications Board, Division on Engineering and Physical Sciences, National Research Council of the National Academies. The National Academies Press, Washington, DC. 2007.
19. Peter Kriens. *JPMS, The Sequel*. http://aqute.biz/2017/06/14/jpms-the-sequel.html
20. Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE'16), Seattle, WA, USA, 2016, pp. 373–383.
21. Nuno Macedo, Alcino Cunha and Tiago Guimaraes. Exploring Scenario Exploration. *Fundamental Approaches to Software Engineering* (FASE 2015), A. Egyed and I. Schaefer, eds. Lecture Notes in Computer Science, Vol 9033. Springer, Berlin, Heidelberg.
22. Nuno Macedo, Alcino Cunha and Eduardo Pessoa. Exploiting Partial Knowledge for Efficient Model Analysis. *15th International Symposium on Automated Technology for Verification and Analysis* (ATVA'17), pp 344-362. Springer, 2017.
23. Baoluo Meng, Andrew Reynolds, Cesare Tinelli and Clark Barrett. Relational Constraint Solving in SMT. *26th International Conference on Automated Deduction* (CADE '17) (Leonardo de Moura, ed.), Springer, Vol. 10395, Gothenburg, Sweden, 2017.
24. Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang and Daniel Jackson. Alloy*: a general-purpose higher-order relational constraint solver. *Formal Methods in System Design*, 2017, pp.1–32.
25. Vajih Montaghami and Derek Rayside. Extending alloy with partial instances. *Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z* (ABZ'12), 2012, pp. 122–135.
26. Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, Shriram Krishnamurthi. The Margrave Tool for Firewall Analysis. *24th USENIX Large Installation System Administration Conference*, San Jose, CA, 2010.
27. Timothy Nelson, Natasha Danas, Daniel J. Dougherty and Shriram Krishnamurthi. The Power of Why and Why Not: Enriching Scenario Exploration with Provenance. *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2017.
28. Timothy Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler and Shriram Krishnamurthi. *Aluminum: Principled Scenario Exploration through Minimality*. *International Conference on Software Engineering*, 2013.
29. Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos Made EPR: Decidable Reasoning about Distributed Protocols. *Object-Oriented Programming, Systems, Languages & Applications* (OOPSLA 2017), Vancouver, 2017.
30. Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst and Jonathan Jacky. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. *Computer Aided Verification* (CAV 2016). Lecture Notes in Computer Science, Vol. 9780, Springer.
31. German Regis, Cesar Cornejo, Simon Gutierrez Brida, Mariano Politano, Fernando Raverta, Pablo Ponzio, Nazareno Aguirre, Juan Pablo Galeotti and Marcelo Frias. DynAlloy Analyzer: A Tool for the Specification and Analysis of Alloy Models with Dynamic Behaviour. *11th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2017). ACM, New York, NY, USA, pp. 969–973.
32. John Michael Spivey. *The Z Notation: A reference manual* (2nd ed.), Prentice Hall, 1992.
33. Ion Stoica, Robert Morris, David Liben-Nowell, David R. 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 (TON), Vol. 11, No. 1 (2003): pp.17–32.
34. Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'07), Braga, Portugal, 2007, pp. 632–647.
35. Caroline Trippel, Daniel Lustig and Margaret Martonosi. *MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols*. arXiv:1802.03802, February 2018.
36. University of Washington. *PLSE Neutrons*. http:neutrons.uwplse.org/
37. W. Visser, K. Havelund, G. Brat, S.-J. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), April 2003.
38. Charles Wallace. Learning Discrete Structures Interactively with Alloy. *49th ACM Technical Symposium on Computer Science Education*, Baltimore, Maryland, USA.February 21–24, 2018, pp. 1051–1051.
39. Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley, 1999.
40. Hillel Wayne. Personal blog. https://www.hillelwayne.com
41. John Wickerson, Mark Batty, Tyler Sorensen and George A. Constantinides. Automatically comparing memory consistency models. *44th ACM SIGPLAN Symposium on Principles of Programming Languages* (POPL 2017),Paris, France, 2017, pp. 190–204.
42. Pamela Zave. A practical comparison of Alloy and Spin. *Formal Aspects of Computing*, Vol. 27: 239–253, 2015.
43. Pamela Zave. Reasoning about identifier spaces: How to make Chord correct. *IEEE Transactions on Software Engineering*, 43(12):1144–1156, December 2017.