Safe Composition of Web Communication Protocols for Extensible Edge Services *

Adam D. Bradley Azer Bestavros Assaf J. Kfoury

Computer Science Department
Boston University
Boston, MA 02215
{artdodge,best,kfoury}@cs.bu.edu

Abstract

As new and complex multi-party edge services are deployed on the Internet, application-layer protocols with complex communication models and event dependencies are increasingly being specified and adopted. To ensure that such protocols (and compositions thereof with existing protocols) do not result in undesirable behaviors (*e.g.*, deadlocks), a methodology is desirable for the automated checking of the "safety" of these protocols. In this paper, we present ingredients of such a methodology. Specifically, we show how SPIN, a tool from the formal systems verification community, can be used to quickly identify problematic behaviors of application-layer protocols with non-trivial communication models—such as HTTP with the addition of the "100 Continue" mechanism. As a case study, we examine several versions of the specification for the Continue mechanism; our experiments mechanically uncover multi-version interoperability problems, including some which motivated revisions of HTTP/1.1 and some which persist even in the current version of the protocol. We develop relations for describing arbitrarily large compositions of HTTP proxies using finite models, and also discuss the broader applicability of these techniques to open internet protocol development.

Keywords: Formal verification, HTTP, Interoperability, Model checking, Protocol composition.

1 Introduction

Stateful multi-party protocols can be notoriously difficult to get right, and their design and implementation is a process demanding careful thought. The evolution of the HTTP protocol is a case in point. While the original formulations of the HTTP protocol were truly stateless and thus relatively easy to implement, the addition of the multi-stage `100 Continue` mechanism to HTTP/1.1 [14] implicitly introduced several "states" to the behavior of clients, servers, and intermediaries. Not surprisingly, an ambiguity was discovered in the handling of these states with respect to intermediaries which could, under some "correct" interpretations, lead to a deadlock state among conforming implementations of HTTP/1.1 (RFC2068) and HTTP/1.0 (RFC1945) [17].

For years, analogous problems have been commonplace in the design of lower-level distributed protocols; mastering all the nuances of handshaking, rendezvous, mutual exclusion, leader election, and flow control in such a way as to guarantee correct, deadlock-free, work-accomplishing behavior requires very careful thought, and hardening the specifications and implementations of these protocols to deal with misbehaving or potentially hostile peers remains a difficult problem for engineers at all layers of the stack. These problems arise in such settings even without the complex multi-version interoperability goals placed upon HTTP/1.1 and its revisions; while this may be comforting to those who have incorrectly designed or implemented protocols in the past, it is discouraging to those of us who would like to see HTTP reach a fully-correct "stable" state.

The formal methods and protocol verification communities have been mindful of these problems for some time, and have developed quite usable tools which allow us to examine and address these problems in a rigorous, unambiguous way [8]. The PROMELA language and the accompanying SPIN verifier [9], for example, have proven particularly well-suited to describing and analyzing non-deterministic sets of acceptable behaviors for responsive (event-driven) systems and protocols; such a view of the world is easily applied to multi-state application layer protocols to facilitate analysis of their behaviors and assessment of their correctness in interacting both with friendly and malevolent peers.

In this paper, we bring some of these tools and processes to bear upon the `100 Continue` feature of the HTTP protocol. In so doing, we identified instances of failure modes under client-proxy-server operating arrangements– some well-known [17] and some apparently not. We

*This research was supported in part by NSF (awards ANI-9986397, ANI-0095988, CCR-9988529, and ITR-0113193) and U.S. Department of Education (GAANN Fellowship)

have also proceeded to deduce relations among the set of such arrangements which identify equivalence classes of deadlock-prone or non-deadlock-prone situations, and deduce two patterns of agents whose presence indicates a deadlock-prone arrangement of HTTP agents. All of these properties form important components of our long-term research agenda, which deals with programming composable distributed applications which may use protocols like HTTP as part of their infrastructure.

Internet Flows as First-Class Values: Programming new services in the Internet currently suffers from the same lack of organizing principles as did programming of stand-alone computers some thirty years ago. The Web community's experiences in relationship to the evolution of HTTP–as well as findings we present in this paper–underscore this state of affairs. Primeval programming languages were expressive but unwieldy; programming language technology improved through better understanding of useful abstraction mechanisms for controlling computational processes. We would like to see the same kinds of improvements find their way into the programming of distributed Internet services. In so doing, we believe that one property of this improvement is the promotion of network flows to *first-class values*, that is, objects which can be created, named, shared, assigned, and operated upon. This requires new paradigms for those programming or creating new services; it also demands more rigorous abstraction of the services and infrastructures which will support those distributed programs. We are currently working toward developing these and other mechanisms and integrating them into a network programming workbench environment we call NETBENCH.

The world we envision NETBENCH operating in is one with vast multitudes of widely varied network applications and services, each with unique needs in terms of resources, input and output formats, reachability, and other communication parameters. The problem of actually composing these services in a manner that preserves all of the important properties is profoundly difficult: How do we ensure that the properties of each layer of encapsulation preserve the requirements of the encapsulated flows? How do we ensure that gateways can properly convolve wildly different communication models represented by each of their protocols? How do we ensure that certain meta-data properties will be preserved across gateways and through caches? How does one ensure that the required network resources can be allocated, perhaps probabilistically, between herself and the series of service points which are cooperating to produce the output?

Type systems have proven to be a powerful mechanism for verifying many desirable properties of programs for stand-alone systems.[1] We believe that they can also capture many essential correctness properties for the composition of networked services. By forcing a strong typing system onto agents and flows in the network, we can build up a library of strongly typed operations which can be performed upon those flows (composition: $[A] + [B] \rightarrow [A + B]$), legal "casting" operations (an HTTP/1.0 client can safely speak with an HTTP/1.1 server), polymorphic operations (tunneling any TCP flow via SSL), and even build type inference systems which could mechanically deduce the need for additional agents in a path.

Of course, such a type system depends upon a rigorously correct knowledge of the underlying systems and technologies which are being typed, which leads us to the specific contributions of this paper.

Paper Contributions and Outline: The methodology we employ in this paper to analyze the HTTP protocol provides us with important ingredients of NETBENCH–specifically: (1) it shows us how certain instantiations of HTTP arrangements can be unsafe, and must be rendered unsafe by a type system or type inference engine, for example; (2) it provides us with a set of bidirectional equivalences between topologies of agents can be applied to "reduce" a system to make its analysis more tractable, or to "stretch" a system to include more agents without altering a desired property; (3) it discusses a generalizable method for testing the safety and correctness of other properties which affect the behavior of protocol agents; and (4) it provides us with a hands-on example of how to rigorously approach the establishment of such properties for use in a strong type system.

The primary focus of this paper is on the applicability of formal methods to the verification of the correctness and interoperability of the HTTP protocol's revisions in all permutations of roles (client, proxy, server) and compositions thereof with respect to the `100 Continue` feature. We do not focus at great length upon the nuances of disambiguating the RFCs;[2] where we have made simplifying assumptions or omissions, we briefly discuss why.

While much of the work in this paper was done manually, it is evident that many of the tasks and inferences made are plausible candidates for mechanical analysis and deduction; our hope is that the lessons learned will lead us to algorithms and results toward that end, and some such results will soon be forthcoming [3].

The remainder of this paper is organized as follows. We begin with an overview of the capabilities that a formal methods toolset (like SPIN) offers to the application-layer protocol design and implementation communities. We then present, as a case study, some results of our examination of the interoperability of multiple revisions of the HTTP

[1] Type systems provide one methodology, one of several from the Programming Languages research community, which can be used to formally show that certain safety properties are met. Finite-model checking is another such methodology.

[2] Neither do we address, at any length, the issue of validating actual implementations of RFCs against the RFCs themselves or against models thereof.

protocol [1, 5, 6]. We follow that with a set of rules that we developed for curtailing the state space of HTTP protocol compositions. We conclude the paper with a summary of our findings.

2 Benefits of Formal Verification

Since formalization and the application of formal methods is sometimes shunned by members of the hacking and software engineering communities, it is worth re-stating some of the benefits which they have to offer.

Disambiguation: While RFCs and related standards documents tend to be fairly unambiguous when it comes to syntax and grammar, the specification of semantics in prose lends itself to incompleteness in its coverage of all possible scenarios or inputs. While in some regards this is desirable as it allows great freedom to implementors, it can at the same time leave the door open to unintended interpretations which may aversely affect the behavior of the system.

Formalizing the communication behavior of each agent's role under a protocol using some rigorous technical representation forces the protocol designers to think concretely about the sequences of events each agent may encounter and what the permissible set of responses to each should be; said another way, disambiguation causes classes of ambiguities in the protocol's specification to be weeded out or allowed to remain *by design*.

Correctness: We would like to prove that correct implementations of a protocol acting in all combinations of roles are *well behaved*, by which we mean that the rules of the protocol prevent the system from entering undesirable states such as deadlock (all agents blocked waiting for others to act) or livelock (agents interacting in a way that produces no "progress").

Interoperability: Incremental improvements and enhancements to protocols are the rule of thumb for the Internet at the application layer. As such, an important design goal for each incremental version of a protocol is backward-compatibility with implementations of previous versions of the protocol, in all roles for which they may appear. By interoperable we mean both that the system is able to accomplish useful work (*i.e.*, any bootstrap problems are handled gracefully) and that the system is well behaved in the sense described above. Ideally, an implementation of a new version of a protocol should be able to replace an older one in any single role under any given arrangement of agents and the usefulness and correctness of the system should not be disturbed.

Hardening: A pressing concern in the wild is the issue of what happens when our well-crafted and theoretically correct and interoperable protocol has to interact with the great unknown of poorly written, misbehaving, or even malicious peers. We may wish to ensure that we interoperate with certain particular deviant implementations, which we can capture by modeling their aberrations and applying the same interoperability testing regimen discussed above. However, this is a more general problem: do the requirements of our protocol prevent some sequence of inputs from causing an agent to behave poorly, or to have its resources in some way monopolized? Asked another way, does the specification of the protocol require that implementations be sufficiently *hardened* against an arbitrary and potentially hostile world?

A formal verification system allows us to attach models of agents to what is called a "maximal automaton", an agent which feeds all possible sequences of input to our model. The verification system then examines the product of the agent's model with the maximal automaton and tests all possible execution paths for progress or graceful termination. This notion is not unique to the protocol verification area; for example, an I/O automata [16] is said to be "input enabled" if it is able, in all input-accepting states, to transition (possibly to the same state or to a failure/termination state) in response to any input value, thus "hardening" it against all possible inputs.

Implementation Conformity: While not yet a perfected art, there has been much work toward low-overhead integration of model building and model checking with software engineering processes. Given a relatively stable software architecture and not-too-rapidly moving set of interfaces, it is proposed that by applying "slicing" rules to source code a verification system can extract only that information relevant to the properties and behaviors being modeled, then regularly build and re-validate models from source and check them for the desired correctness, interoperability, and hardening properties discussed above.

The prospect of very-low-overhead tools of this kind finding their way into the development process, particularly for system software, is certainly alluring. That the research community is generating such tools in conjunction not only with academic and experimental languages, but with such favorites as C [11, 12, 10] and Java [4, 7], is cause for hope; this could constitute a much-needed improvement the the state of the art if [13] is any indication.

3 Verifying HTTP: A Case Study

In this section we present our methodology for evaluating the safety of composition of Web communication protocols in NETBENCH, by working through a case study that assesses the correctness and interoperability of the various revisions of the HTTP specification with respect to the `100 Continue` feature. The exploration and analyses

we present for this case study will serve as a template for more such work in the future with other application-layer protocols.

3.1 A Propos

In HTTP/1.0, all transactions had a very simple and stateless communication model: A client would send a whole request, *i.e.*, a request line, a set of headers, and an optional request entity; The server, after receiving the whole request, would respond with a status line, a set of headers, and an optional response header.

One of the desired features for the 1.1 revision of the protocol was the ability for clients to avoid transmitting very large entities with their requests if the end result of the transaction was to be some simple failure independent of the content of the document (such as an authentication failure or temporary server condition) [14]. Conceptually, this mirrors conditional operations (such as the `If-Modified-Since` header) which allow a response entity to be suppressed if its transmission is deemed unnecessary.

This is was done in RFC2068 by allowing clients to pause before sending the optional request entity; the server could then send a response with an error code in the status line, informing the client that the request would fail and the request entity should not be sent; alternately, the server could send a `100 Continue` response, which tells the client to proceed with sending the request entity (although it does not guarantee that the final response might not still be some error condition).

While the original specification of the continuation mechanism (the `100 Continue` response header, as governed by [5, §8.2 and §10.1.1]) was clearly sound with respect to the simple client-server cases, there was ambiguity as to the correct behaviors of intermediary proxies; compelling arguments could be made that the RFC's language recommended, required, or suggested that the mechanism be applied either hop-by-hop or end-to-end with respect to a chain of proxies. Under at least one of these interpretations, certain combinations of correctly implemented components in the client-proxy-server chain were prone to deadlock [17]. This problem was addressed in the next public revision of the spec (RFC2616) by the introduction of the `Expect` mechanism [6, §8.2.3] and the clarification of the semantics of `100 Continue` [6, §10.1.1] with respect to proxies. Given that many existing 1.1 implementations conformed to various interpretations of the earlier version of the spec, it was decided that RFC2616 should also include a number of heuristics to try to ensure correct interoperation with those implementations.[3] This quagmire of protocol versions, specification versions, special-case interoperability rules, and the set of possible combinations of revisions in the different roles, makes it very difficult to say anything with certainty about the correctness and interoperability of the specification; we can say that it seems *empirically* to be correct, or perhaps that it is even *arguably* correct, but not that it is *provably* so.

As a case study in the application of formal methods to the problems of a protocol's correctness, interoperability, and hardening, we used the SPIN tool [9] from Bell Labs to construct and verify models of the expect/continue behavior of clients, proxies, and servers conforming to RFC1945 (HTTP/1.0), multiple interpretations of RFC2068 (obsolete HTTP/1.1), and RFC2616 (HTTP/1.1).[4]

3.2 PROMELA and SPIN

PROMELA, the PROcess (or PROtocol) MEta LAnguage, is the input language to the SPIN verifier [9]. PROMELA is a non-deterministic guarded command language, which means (informally) that it can represent sets of simultaneously valid reactions to the state of and inputs to a process, and that a process can "sleep" on boolean expressions or external events. Mastering PROMELA's syntax and grammar is a straightforward exercise for anyone familiar with imperative programming and the syntax of C; mastering its semantics simply requires a familiarity with the event-driven or state-machine-driven programming techniques.

PROMELA provides a set of abstractions convenient to modeling local and distributed protocols and systems, including dynamically creatable processes with access to both local and global variables, "dummy" variables which do not effect the analyzed state space of the model, finite-length message queues, and a set of send, receive, and poll-type operators which can operate on those queues.

When SPIN is run on a PROMELA program, it begins by transforming each process description into a finite state machine. After performing some analysis and state reductions, it performs a depth-first search of execution paths of the whole program, searching for cases which violate a set of constraints. The easiest property to test for is deadlock; if the system can reach a state in which one or more processes have not terminated and no process has a runnable instruction (*i.e.*, all processes are "asleep" waiting for predicates to change or events to occur), then the system is deadlock-prone, and the execution path leading to that instance of deadlock is output to the user.[5] Since depth-first search can easily lead to extremely long exemplars, the system can then continue the search while bounding the depth, looking

[3] This naturally raises several fair administrative and technical questions. For example: Was RFC2068 really ready to be released as a Standards-Track RFC? Given the numerous issues raised in post-RFC2068 drafts and the number of heuristics needed to interoperate with "old" HTTP/1.1 agents, would it not have made more technical sense to name the RFC2616 revision HTTP/1.2?

[4] None of the current errata for RFC2616 listed at `http://purl.org/NET/http-errata` pertain to the expect/continue mechanism, so they are not addressed

[5] Another trivially testable property is *progress*, the property that any infinite execution of a process includes infinitely many executions of particular *progress markers*. Absence of this property is indicative of *livelock*.

for shorter constraint-violating exemplars until a shortest one is found. If no error cases are found, the system reports success to the user.

3.3 PROMELA Model of HTTP Expectation and Continuation

The key to building useful and analyzable models is to abstract away enough details to make the models a manageable size while retaining enough detail to be meaningful and reflective of the underlying processes and behaviors.

To represent the basic units of communication among HTTP agents, our models transmit and receive six types of messages (PROMELA "`mtype`"s):

request - corresponds to the "Request" grammar in [6, §5] up to and including the `CRLF`, but excluding the `[ message-body ]`. This message carries a parameter structure with the following fields:

- **version** - Version of the agent sending this message. HTTP_09, HTTP_10, HTTP_11, or some higher value.
- **hasentity** - Boolean indicator of whether a `message-body` will follow the request. This is an abstraction of the rules for inclusion found in [6, §4.3 ¶5] *etc*.
- **expect100** - Boolean indicator of the presence or absence of the `100-continue` in the `Expect` header. Only set explicitly by RFC2616 client implementations, although it may be "passed on" by RFC1945 proxies.
- **close** - Boolean flag that the client is requesting the connection be closed after this request is completed.

response - corresponds to the "Response" grammar in [6, §6] up to and including the `CRLF`, but excluding the `[ message-body ]`. This message carries a parameter structure with the following fields:

- **version** - As above.
- **hasentity** - As above; abstraction for the rules in [6, §4.3 ¶6] *etc*.
- **close** - Boolean flag indicating that the server will close the connection when its response has been completely sent.

continue - corresponds to a "Response" with a status code of 100 ("Continue") and subject to the other restrictions of [6, §10.1, §10.1.1].[6]

entitypiece - a block of bytes constituting part of a `message-body`.

entityend - a block of bytes marking the end of a `message-body`. This message is only sent immediately after one or more **entitypiece**s, and is an abstraction for the various mechanisms which can be used to delineate a `message-body` (`Content-Length`, the `chunked` Transfer-Encoding, the `multipart/byteranges` Content-Type, etc).

eof - a "close" event, in which a party to the connection explicitly shuts down the transport.

This data is sufficient to control all of the behaviors surrounding Continuation described in [5, §8.2, §10.1, §10.1.1] and [6, §8.2.3, §10.1, §10.1.1, §14.20]. We do not currently model the backoff-and-retry mechanism discussed in [6, §8.2.4].

3.4 Implementations of Agents

For each role defined by HTTP (client, proxy, server) and for each revision of the spec (RFC 1945, 2068, and 2616), a PROMELA process model (or `proctype`) was created for agents acting as that role/revision. The naming convention is *role-rfc#[-variant]*; for example, an RFC2616 (HTTP/1.1) server is called *server-2616*, and the hop-by-hop interpretation of an RFC2068 proxy is *proxy-2068-hbh*.

All agents (including RFC1945 agents) are assumed to use persistent connections; this simplification allows us to exhaust the space of transaction sequences without having to spawn multiple agents, and also naturally emulates the "upstream version cache" feature appearing in HTTP/1.1. Pipelining is not implemented because it has no effect (causally) upon any agent's behavior (apart from a more general issue with clients using blocking write operations in the presence of finite buffer space).

Table 1 lists all of the models, the spec they conform to, the role they act in, their approximate size in lines of PROMELA code (LOC), and comments concerning their interpretation of the specifications (as discussed below). The PROMELA code is quite readable, and is available from the author's web site[7].

The HTTP/1.0 proxy is modeled as a HTTP/1.0 client attached to a HTTP/1.0 server; it waits for an entire request (entity included) to arrive, then forwards the whole request to the next upstream agent, and similarly for the response. In the more general case, an HTTP/1.0 proxy could choose to pass through request and response entities progressively as it receives them, but that behavior would be causally indistinguishable to its peers from the whole-entity method we employ.

[6] To correctly implement [6, §8.2.3 ¶5], we would also add a parameter indicating from which node this message originated. Unfortunately, it is unclear that a client could actually unambiguously deduce whether a message comes from an origin server or not, as discussed below in footnote10; hence, we omit this from our model.

[7] Code and other information can be found at `http://cs-people.bu.edu/artdodge/research/httpverify/`

Name	Models	Role	LOC	Comments
client-1945	RFC1945 (1.0)	Client	60	supports keepalive; trivial
client-2068	RFC2068 (1.1)	Client	110	
client-2616	RFC2616 (1.1)	Client	110	
server-1945	RFC1945 (1.0)	Server	60	supports keepalive; trivial
server-2068	RFC2068 (1.1)	Server	90	
server-2616	RFC2616 (1.1)	Server	150	MAY in [6, §8.2.3 ¶8] is "MUST after timeout"
server-2616-may	RFC2616 (1.1)	Server	150	implements [6, §8.2.3 ¶8] as MAY
proxy-1945	RFC1945 (1.0)	Proxy	115	buffers whole requests/responses; supports keepalive; MAY pass thru "expect-100"
proxy-2068-e2e	RFC2068 (1.1)	Proxy	160	end-to-end continue mechanism
proxy-2068-hbh	RFC2068 (1.1)	Proxy	170	hop-by-hop continue mechanism
proxy-2068-hybrid	RFC2068 (1.1)	Proxy	280	selects HBH or E2E randomly per request
proxy-2616	RFC2616 (1.1)	Proxy	150	
proxy-2616-fixed	RFC2616 (1.1)	Proxy	155	Fixes a potential deadlock case (see Section 3.7)

Table 1: HTTP Agent Models

With regard to the behavior of proxies, RFC2068 allows several conflicting interpretations; to deal with this, we created several variations on the *proxy-2068* model, differentiated by a suffix added to the name. The three RFC2068 proxy models are:

proxy-2068-e2e - Interprets continuation as an end-to-end mechanism, whereby the proxy will not ask the client to "Continue" until its upstream agent has asked it to "Continue"

proxy-2068-hbh - Interprets continuation as a hop-by-hop mechanism; the proxy tells the client to continue, reads its `message-body`, then forwards the request upstream and waits for a "Continue" message before forwarding the `message-body`.

proxy-2068-hybrid - As each request arrives, chooses randomly between the *-e2e* and *-hbh* interpretations. This is the general case of a 2068 proxy, meaning a successful validation against it provides us with the strongest result; unfortunately, it is also (by far) the most computationally expensive agent to model. Unless otherwise noted, this model is always used when an RFC2068 proxy is called for, as it completely subsumes the behavior of the other two.

We have made *server-2616* always employ the interoperability clause in [6, §8.2.3 ¶8] when the system encounters a timeout. While the specification makes this behavior a MAY, allowing the model to omit it introduces obvious potential deadlock cases when interacting with RFC2068 downstream peers. While not expounded further in this paper, we convinced ourselves of this property by replacing *server-2616* agents in several cleanly validated arrangements with a model we call *server-2616-may* which can choose arbitrarily to omit this behavior, and found the same deadlocks arising as in arrangements with *server-1945* agents. This is an intuitively obvious result, since the non-MAY server behavior when interacting with an RFC2068 client (not sending a `Continue` and waiting arbitrarily long for the request `message-body`) is indistinguishable from *server-1945*'s behavior with the exception that downstream nodes may have previously convinced themselves that a `Continue` message will be produced by this server.

3.5 Validation Cases

To prove the *correctness* of HTTP/1.1 (RFC2616), we need to verify that all client-server and client-proxy-server combinations of RFC2616 agents can be validated by the SPIN system. We begin by verifying that the simple client-server case and the client-proxy-server case are both correct (deadlock-free). From there we need to somehow convince ourselves that longer (and perhaps arbitrarily so) series of intervening proxies are also correct by the same criteria, as explicitly modeling an arbitrarily long proxy chain is clearly not possible.

Gaining confidence in the *interoperability* property is a much more involved process; for a client-proxy-server architecture like HTTP, a trivial approach requires us to verify as many as $C \times S \times (\sum_{i=0}^{N} P^i)$ arrangements of agents, where C is the number of client models (three), P is the number of proxy models (three), S is the number of server models (three), and N is a heuristic bound on the length of the proxy chain. Once again, the number of arrangements to be checked quickly becomes unwieldy as N grows, and verifying all arrangements when $N = \infty$ is not possible with finite computation, so other means are required to reason about longer chains.

	Client	Proxy	Server	Result	Comments
2.1	1945	all	all	clean	
2.2	all	1945	all	clean	
2.3	all	all	2068,2616	clean	
2.4	all	all	1945	deadlock	
2.5	2068	2068-hybrid	1945	deadlock	
2.6	2068	2068-e2e	1945	deadlock	guilty for 2.5
2.7	2068	2068-hbh	1945	clean	innocent of 2.5
2.8	2068	2616	1945	deadlock	
2.9	2616	2068-hybrid	1945	deadlock	
2.10	2616	2068-e2e	1945	clean	innocent of 2.9?
2.11	2616	2068-hbh	1945	clean	innocent of 2.9?
2.12	2616	2616	1945	clean	

Table 2: Experiments to Localize Interoperability Problems for $P = 1$

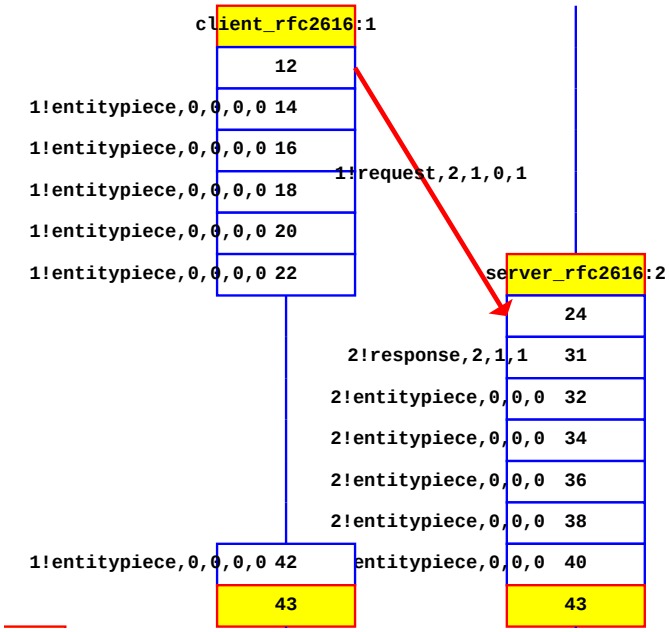

Figure 1: MSC for Buffer Write Deadlock

3.6 Correctness Results

To our surprise, when testing the *client-2616→server-2616* arrangement SPIN promptly returned a very short deadlock case in which both the client and server end up sleeping in write operations, mutually waiting for buffer space to become available. This case arises as follows: The client then sends a request, but does not set the `Expect: 100-continue` header. The server receives the request and decides to reject it, and thus immediately begins sending its response and the entity that follows. Meanwhile, the client has begun to send the request `message-body` without waiting to hear from the server.[8] This situation is illustrated by the message sequence chart (MSC) in Figure 1; step 24 is the only "receive" that is executed; consequently, the message queues (each with a capacity of 5) both fill, causing both processes to block in their write operations in steps 40 and 42. In practice, this would commonly require the request `message-body` to exceed 128KB (64KB of send buffer at the client and 64KB of receive buffer at the server), and the response headers and `message-body` to likewise exceed 128KB. It is not surprising that this error would rarely if ever occur in practice; while it is becoming common practice to submit images in requests (which can easily exceed 128KB), error responses still tend to be fairly small in most cases.

In an ideal world, this deadlock would be purely a defensive engineering (*i.e.* security) concern; however, the more involved communication model of HTTP/1.1 allows it to arise in totally benevolent environments between conforming implementations. If an implementation of HTTP/1.1 is susceptible to this deadlock under benevolent conditions, then it is also possible for a malevolent peer to capitalize upon it to produce a degradation-of-service attack upon a server by causing it to unproductively consume large amounts of outbound buffer space. Thus, in this case, good closed-system engineering and hardening of the system to interact with the open-system world are coincidental and agreeing goals.

To remove this deadlock from all of our models, we changed a macro which produced arbitrarily long entities (both for requests and responses) to produce short fixed-length ones (one **entitypiece** messages followed by an **entityend**), and we enlarged the buffers in each direction so that the entire modeled sequence of messages constituting any client's or server's part in the most involved of transactions could fit comfortably in the buffers.

[8] This situation can also arise if the client sets the `Expect:100-continue` header but elects not to wait for a response to begin sending; this is allowable behaviors under most conditions in RFC2616.

client→*proxy*	→*server* models		
	1945	2068	2616
1945-1945	clean [2.1]		
1945-2068			
1945-2616			
2068-1945	clean [2.2]		
2068-2068	deadlock [2.5]	clean [2.3]	
2068-2616	deadlock [2.8]		
2616-1945	clean [2.2]		
2616-2068	deadlock [2.9]	clean [2.3]	
2616-2616	clean [2.12]		

Table 3: Safety of all *client*→*proxy*→*server* Permutations

Having removed the buffering deadlock from our models, we found that reasonably short RFC2616-only arrangements were all verifiably correct. The verification of the first three arrangements ($N = 0, 1, 2$) required less than 30 seconds of CPU time.[9]

3.7 Interoperability Results

SPIN required only a few seconds to test all nine client-server cases, As none of these cases gives rise to any deadlock conditions, we are confident that all three revisions of HTTP interoperate gracefully in simple client-server arrangements.

Searching through the client-proxy-server cases is a more involved process; the series of experiments is listed in Table 2, and the resulting interoperability matrix given in Table 3. Wherever the word "all" appears in Table 2, it reflects a run of the verifier in which all three principal revisions of that agent are tested in that role; similarly, where multiple models are listed, they are both tested explicitly by that experiment's model. The result is either "clean" (indicating a successful validation of *all* members of the described set of arrangements) or "deadlock" (meaning *at least one* of the arrangements described is prone to a deadlock condition). Each result in Table 3 includes a reference to one experiment in the former which proves it (several cells are actually proven by multiple experiments). The experiments used to produce the table required roughly 15 minutes of CPU time; the process of choosing a testing strategy and setting up the experiments was by far the clock-time bottleneck on the verification process.

Classic 1.1/1.1/1.0 Deadlock: The first class of deadlock detected in experiments 2.5, 2.6 and 2.7 are reflected by the example MSC shown in Figure 2. These cases were identified in 1997 on the HTTP-WG mailing list [17], and we refer to them as the "classic" continuation deadlock, in

[9]Timing results for experiments discussed in this paper were acquired using a Quad Pentium II 450MHz system with 2GB of RAM; initially, the models were developed and the $N \in \{0, 1\}$ arrangements all tested on a modest 400MHz Mobile Pentium II laptop with 128MB of RAM.

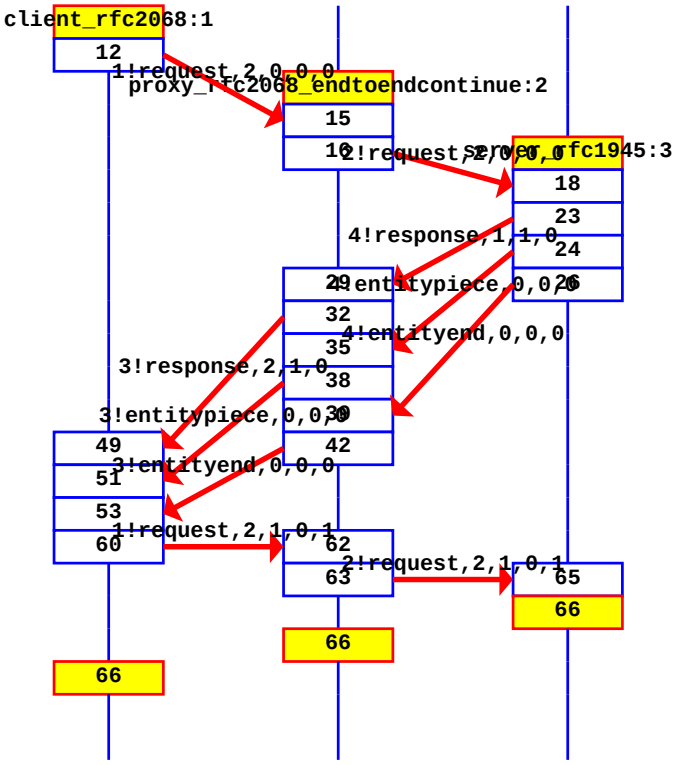

Figure 2: Example MSC for Deadlock of 1.1 Client, 2068 Proxy, 1.0 Server

which an end-to-end interpretation at the proxy has no rule to cause it to balk when it cannot expect its upstream agent to provide a `100 Continue` message and has no compulsion to initiate its own (or alternatively, an error message). Note that an RFC2068 client will only wait for a `100 Continue` if it believes it is interacting with an HTTP/1.1 upstream agent; we model the recommended version cache by simply having each agent remember the version numbers of its peers for the life of a persistent connection. This is why a simple and successful request is completed in the MSC before the actual deadlocking request is made; in the wild, the deadlock could just as easily arise in the first request of a persistent connection if the version cache values are in place.

Deadlock Involving an RFC2616 Proxy: Experiment 2.8 took us by surprise, as it is virtually identical to the "classic" deadlock mentioned above, except that it includes an RFC2616 proxy (the revision that was supposed to interoperate gracefully with older revisions)! The shortest example MSC is virtually identical to Figure 2. Following the warming of the version caches, the client, believing it is communicating with an HTTP/1.1 server which will provide it with a `100 Continue` signal, sends request head-

ers indicating a `message-body` will follow. Because it implements RFC2068 it does not know about the `Expect` header, so it does not send it.

The proxy is simply unable to resolve this situation correctly using the rules of RFC2616; it knows that the upstream server is HTTP/1.0 and neither understands `Expect` nor will it provide `100 Continue`. However, while RFC2616 requires that in such a case a request to a proxy including a `100-continue` expectation be answered with an error response status of 417 (Expectation Failed) [6, §8.2.3 ¶13,14], this requirement does not apply to requests which have no `Expect` header, as the request from the *client-2068* agent does not.

While RFC2068 does not say that clients in general MUST wait for a 100 from an upstream server, it is required under the "retry" rules if the upstream agent is known to be HTTP/1.1, and nowhere in that spec are clients required to bound the time they will wait for a Continue message.

This deadlock can be resolved by altering the spec and borrowing an idea from a compatibility rule for origin servers in [6, §8.2.3 ¶8] and allowing (requiring?) proxies to initiate their own `100 Continue` messages when they receive an HTTP/1.1 PUT or POST request without a `100-continue` expectation token and know that the next server upstream is HTTP/1.0 or has never sent a `100 Continue` message. We model this behavior using *proxy-2616-fixed*; if it replaces *proxy-2616* in experiment 2.8, that experiment successfully validates without deadlocking.

Hybrid Proxy Deadlock: One interesting deadlock condition arises purely because we use the *-hybrid* model in our experiments; an example of this deadlock is illustrated in Figure 3. Notice how experiment 2.9 reports a potential deadlock, while experiments 2.10 and 2.11 show that replacing the *-hybrid* node with either *-e2e* or *-hbh* leads to a clean validation; this is simply explained by one of the client rules in RFC2616 [6, §8.2.3 ¶5] which (interestingly enough) was added to try to work around such problems. This rule allows the client to wait indefinitely for a server to provide it with a `100 Continue` message if it has received one from that server before; the proxy switching from its *-hbh* persona (which provides `100 Continue` messages autonomously) to its *-e2e* persona (which cannot produce one, nor does it know how to balk at its knowledge that the upstream server is HTTP/1.0).[10]

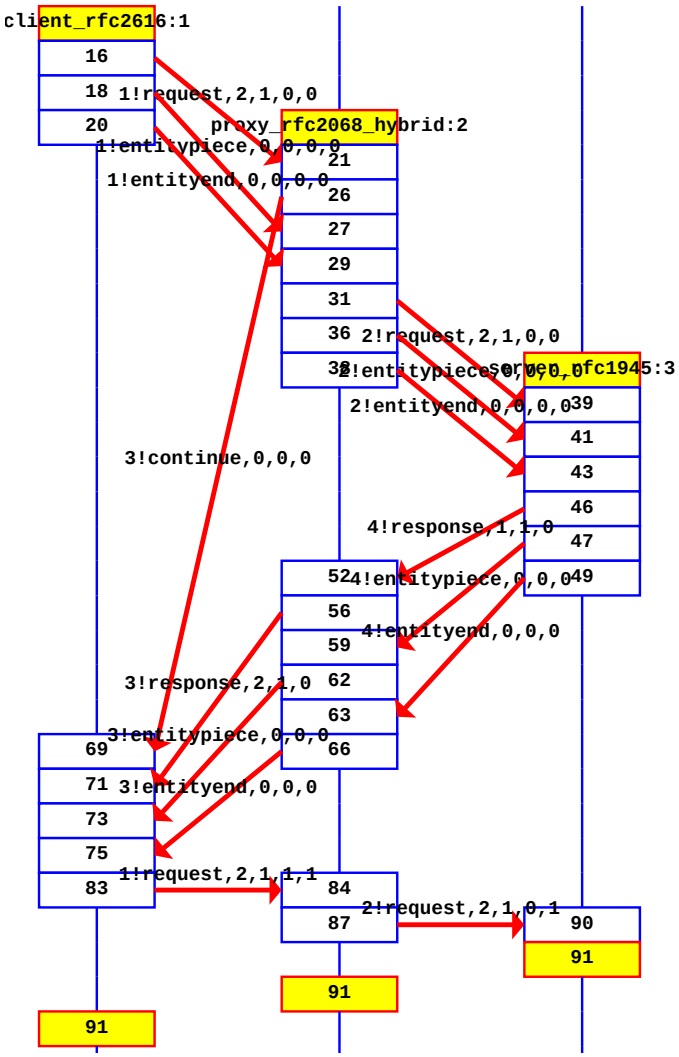

Figure 3: Example MSC for Hybrid Proxy Deadlock

Longer Chains A summary of experiments for client-proxy-proxy-server (*i.e.* $N = 2$) arrangements is presented in Table 4, with the safety matrix of the 81 primary arrangements presented in Table 5 using the same format as above. Table 5 also refers to experiments from Table 2 where the the deadlocks from the two experiments arise for the same causes.

For example, we notice immediately that many $N = 2$ deadlocks are analogous with experiment 2.8, that is, because *proxy-2616* is unable to correctly reconcile the client (*client-2068*)'s lack of a `Expect: 100-continue` header with the proxy's knowledge that the upstream server is HTTP/1.0 and will therefore not provide a `Continue` message. This is a problem which we addressed above with

[10] While the spec is particular about only regarding `100 Continue` messages which actually come from the origin server, it is not clear that a client can correctly disambiguate whether such a message came from the origin server. The only way a client could do so is by comparing the `Via` headers from a `100 Continue` and the following final response. It is not clear that correct proxies will append `Via` headers to `100 Continue` messages, since [6, §10.1] which describes proxy forwarding behavior for 1xx messages says that "There are no required headers for this class of status code"; while it would be a good interpretive step to always add `Via`, making that requirement explicit would help further clarify the spec. Regardless, given this ambiguity, a client must either never wait indefinitely (which violates the spirit of the rule, but is one behavior captured by our models) or wait indefinitely under uncertain conditions (which is also a behavior captured in our model, the selection of which leads to this deadlock).

	Client	Proxy	Proxy	Server	Result
4.1	all	1945	all	all	clean
4.2	1945	all	1945	all	clean
4.3	all of 2068,2616	2068-hybrid	1945	all	deadlock
4.4	2068	2068-e2e	1945	all	deadlock
4.5	2068,2616	2068-hbh	1945	all	clean
4.6	2068	2616	1945	all	deadlock
4.7	2616	2068-e2e, -hbh	1945	all	clean
4.8	2616	2616	1945	all	clean
4.9	all	all	2068, 2616	2616	clean
4.10	all	2068	2616	1945	deadlock
4.11	1945,2616	2616	2616	1945	clean
4.12	2068	2616	2616	1945	deadlock
4.13	2068,2616	2068,2616	2068	1945	deadlock
4.14	1945	2068	2068	1945	deadlock
4.15	1945	2616	2068	1945	clean

Table 4: Localizing Interoperability Problems for $P = 2$

the *proxy-2616-fixed* model.

Identifying such relationships helped in discovering the set of reduction rules and failure classes discussed below.

4 State Space Reduction

In the previous section, we demonstrated how NETBENCH would be able to leverage the SPIN formal verification tool to discover various unsafe behaviors of finitely long protocol (*e.g.*, HTTP) compositions. One of the hurdles that face many tools such as SPIN is the state space explosion problem, which raises concerns as to their scalability for non-toy problems; the extreme instance of this problem is the inability of such tools to validate the (infinite) set of all arbitrarily large systems directly using a finite model. In this section, we show that the application of domain-specific knowledge can mitigate this problem.

Since every deadlock case which appears in the two-proxy experiments is reflective of a deadlock condition already discovered in the single-proxy experiments, and since HTTP's behaviors are all either end-to-end or hop-by-hop, it would seem reasonable to guess that (for these models at least) there are deadlock-prone *patterns* which longer chains could be checked for to determine whether they will be deadlock-prone or not. This leads us to postulate a set of reduction rules, according to which we can organize and partition the infinitely large search space of arrangements into classes which are equivalent with respect to being deadlock-prone or deadlock-safe.

Reductions: Since the set of arrangements needing explicit validation grows exponentially with N, it makes far more sense to talk about the properties of subsequences of agents and build up a description of the set of deadlocking arrangements (or its inversion, the set of safe arrangements). Essentially, we would like to describe a *language* (in the formal sense) of arrangements which fall into either category. Toward that end, we here describe two sets of findings: First are a set of relations called "reductions" which map large sets of arrangements onto smaller sets; Second are *failure patterns*, that is, sequences of agents which describe a deadlock-prone condition in any arrangement (or arrangement reducible to one) which they match.

Through careful study of our interoperability results and our models of the protocol agents, we have been able manually to deduce a set of reductions, some examples of which are given below. These reductions preserve the property of *safety with respect to expectation*; if a given arrangement of agents is deadlock-prone, then any arrangement which is reducible to that one will also be deadlock-prone, and likewise any arrangement it reduces to will be deadlock-prone; the same holds for arrangements which are deadlock-free. Note that most of these reduction can be iteratively and recursively applied; for example, when one reduction allows a chain of agents to reduce to a single agent, that single agent is truly single and qualifies for reduction via other rules which call for single agents.

While we are exploring techniques for deducing these equivalence relationships mechanically rather than manually, that work is beyond the scope of this paper.

1. Our model of a *proxy-1945* is, thanks to restrictions placed upon HTTP/1.1 agents, indistinguishable from a *server-1945* to downstream (toward the client) agents in indistinguishable from a *client-1945* to upstream agents; consequently, the arrangement *x*→*proxy-1945*→*y* verifies if and only if *x*→*server-1945* and *client-1945*→*y* both verify.

2. As a corollary to 1, a series of *proxy-1945* agents is equivalent to a single such agent because each hop is modeled by the *client-1945*→*server-1945* arrangement which is deadlock-free. That single agent can itself then be removed using 1.

client→proxy→proxy	→server models		
	1945	2068	2616
1945-1945-1945	clean [4.1]	clean [4.1]	clean [4.1]
1945-1945-2068	clean [4.1]	clean [4.1]	clean [4.1]
1945-1945-2616	clean [4.1]	clean [4.1]	clean [4.1]
1945-2068-1945	clean [4.2]	clean [4.2]	clean [4.2]
1945-2068-2068	deadlock [4.14, 2.5]	clean [4.9]	clean [4.9]
1945-2068-2616	deadlock [4.10, 2.8]	clean [4.9]	clean [4.9]
1945-2616-1945	clean [4.2]	clean [4.2]	clean [4.2]
1945-2616-2068	clean [4.15]	clean [4.9]	clean [4.9]
1945-2616-2616	clean [4.11]	clean [4.9]	clean [4.9]
2068-1945-1945	clean [4.1]	clean [4.1]	clean [4.1]
2068-1945-2068	clean [4.1]	clean [4.1]	clean [4.1]
2068-1945-2616	clean [4.1]	clean [4.1]	clean [4.1]
2068-2068-1945	deadlock [4.3, 2.5]	deadlock [4.3, 2.5]	deadlock [4.3, 2.5]
2068-2068-2068	deadlock [4.13, 2.5]	clean [4.9]	clean [4.9]
2068-2068-2616	deadlock [4.10, 2.8]	clean [4.9]	clean [4.9]
2068-2616-1945	deadlock [4.6, 2.8]	deadlock [4.6, 2.8]	deadlock [4.6, 2.8]
2068-2616-2068	deadlock [4.13, 2.8]	clean [4.9]	clean [4.9]
2068-2616-2616	deadlock [4.12, 2.8]	clean [4.9]	clean [4.9]
2616-1945-1945	clean [4.1]	clean [4.1]	clean [4.1]
2616-1945-2068	clean [4.1]	clean [4.1]	clean [4.1]
2616-1945-2616	clean [4.1]	clean [4.1]	clean [4.1]
2616-2068-1945	deadlock [4.3, 2.9]	deadlock [4.3, 2.9]	deadlock [4.3, 2.9]
2616-2068-2068	deadlock [4.13, 2.5]	clean [4.9]	clean [4.9]
2616-2068-2616	deadlock [4.10, 2.8]	clean [4.9]	clean [4.9]
2616-2616-1945	clean [4.8]	clean [4.8]	clean [4.8]
2616-2616-2068	deadlock [4.13, 2.9]	clean [4.9]	clean [4.9]
2616-2616-2616	clean [4.11]	clean [4.9]	clean [4.9]

Table 5: Safety of all client→proxy→proxy→server Permutations

3. When a *proxy-2616* immediately follows a *client-2616*, it is equivalent to an arrangement in which the *proxy-2616* is absent.

4. The same can be said of *proxy-2616* when it immediately precedes *server-2616*.

5. A series of *proxy-2616* agents anywhere in the arrangement reduces to a single *proxy-2616*.

6. Unfortunately, arguments like 2 and 5 do not apply directly to *proxy-2068-hybrid*. However, any series of *-hybrid* agents of length greater than 2 is equivalent to one of length 2.

7. Taking a more careful look at the interactions between *client-1945* and an immediately upstream *proxy-2068-hybrid* we find that, to upstream nodes, that particular pair is indistinguishable from a *client-2068*.

Given that all client-server pairs are clean, this set of reduction is already sufficient to explain all but five of the 24 clean validations for the $N = 1$ experiments in Tables 2 and 3 because they are reducible to clean client-server cases ($N = 0$). Those five cases all involve the peculiarities of interactions between RFC2068 and RFC2616-conforming HTTP/1.1 agents.

The above reductions are discussed and justified in some greater depth in [2], which also presents some additional reduction relations; most of the omitted reductions deal only with the *-e2e* and *-hbh* variants of *proxy-2068*.

Failure Patterns: From our interoperability experiments, we have deduced two patterns which can identify which longer chains will and will not be deadlock-prone: if a longer chain can be reduced to a chain containing either of these patterns, it is a deadlock-prone arrangement.

Both of these cases involve interacting with an agent with a behavior equivalent to *server-1945*; Recall that in the wild, this could include RFC2616 servers which do not implement the interoperability MAY (*i.e.*, servers which correspond to the *server-2616-may* model discussed in Section 3.4) and are interacting with an HTTP/1.1 proxy immediately downstream.

1. Any arrangement in which an HTTP/1.1 proxy (*proxy-2068-hybrid* or *proxy-2616*) must interact with an

RFC2068 agent immediately downstream (*client-2068* or *proxy-2068-hybrid*) and a *server-1945* (or equivalent) agent immediately upstream will be deadlock-prone. This was discussed above in connection with experiment 2.8, and also corresponds with experiments 2.5, 4.3, 4.6, 4.10, 4.13, and 4.14.

2. Any arrangement in which a *proxy-2068* must interact with a stream of entirely HTTP/1.1 agents downstream (whether RFC2068 or 2616) and a *server-1945* (or equivalent) agent immediately upstream will be deadlock-prone. This rule corresponds with experiments 2.5, 2.9, 4.3, and 4.13.

More particular failure patterns which distinguish among the *-hybrid*, *-e2e* and *-hbh* variants of *proxy-2068* are straigtforward to derive, but are not presented in this paper as they add little in the way of methodological or intuitive insight; for now, we simply note that under some conditions it is the *-e2e* persona of *-hybrid* which is responsible for the deadlock (in which case substitution by an *-hbh* would alleviate the condition), while under other conditions it is the fact that *-hybrid* agents of any flavor will wait indefinitely without sending the `Expect: 100-continue` flag.

These failure classes, combined with the reduction rules above, are sufficient to explain all of the deadlock cases for the $N \leq 2$ arrangements; all $N = 1$ and $N = 2$ deadlock-prone arrangements match one of the two patterns by way of the application of one or fewer reductions (usually 1; 5 and 7 are also useful).

Indefinitely Long Arrangements In another forthcoming paper [3], we show that these results are not sufficient to characterize the infinite set of all possible arrangements through finitely many model checks. We then show that by adding a particular reduction which only applies to cases of $N \geq 3$, every member of the infinite set of all possible arrangements is reducible to one of 53 models of length $N \leq 4$. While the methodology behind this result is beyond the scope of this paper, we present the additional reduction here for completeness:

8. Consider arrangements containing the sequence *proxy-2068-hybrid*→*proxy-2616*→*proxy-2068-hybrid*. The passive behavior of *proxy-2616* will never initiate a Continue message of its own, neither will it add any expectation to the upstream path which was absent at the downstream *proxy-2068-hybrid*; its behavior is end-to-end, and thus it will never block an inbound message; furthermore, since both *proxy-2068-hybrid* and *proxy-2616* self-identify as HTTP/1.1, it will have no effect upon the perceived versions of messages received by either *proxy-2068-hybrid*. Therefore, this arrangement is equivalent to one in which the middle *proxy-2616* is removed.

Based upon this result and modeling of the necessary $N > 2$ cases, we found that our two failure patterns are sufficient to identify all deadlock-prone arrangements among the 53 irreducible arrangements; since all arrangements of arbitrary length are reducible to members of this set, we have therefore exhaustively partitioned the set of all possible arrangements, and can therefore trivially determine whether any arrangement of arbitrary length is deadlock-prone or deadlock-safe.

5 Model Checking, HTTP, and Web Services

It is natural to ask how generalizable the methods presented in this paper are to other attributes of HTTP, or to other protocol features in general. As alluded to earlier, finite-state model checking has been applied successfully to stateful protocols/protocol features, such as handshaking and leader election. Some features of HTTP naturally lend themselves to this kind of modeling; expectation/continuation and backward-compatible handling of persistent connections are two straightforward examples.

However, much of HTTP/1.1's feature set and parameter space are not actually linked with the behavior of HTTP as such; instead, those headers are used to convey meta-data which governs how the applications which employ HTTP are to handle the data it is used to transport. The "correctness" of these features is not really representable in terms of HTTP itself; rather, it requires some sort of model of the HTTP-utilizing application which can be verified to maintain certain properties or avoid certain sequences of events. (For example, HTTP/1.1 proxy-caching is simply a particular HTTP intermediary application which is largely controlled by the appropriate portions of the spec; one would have to model the internal logic of the cache management code in order to verify that it conforms to the requirements of the RFC.) This observation, that HTTP tries to specify and accomplish several goals at several "layers" simultaneously [15, 18], was one argument which emerged from the HTTP-NG initiative for for decomposing HTTP. Indeed, HTTP/1.1 is difficult to compartmentalize even on a header-by-header basis, as headers such as `Warning` and operating modes such as the `trailers` transfer-encoding affect and are affected by both HTTP-layer and application-layer events. This can make it difficult to assess which features are modelable and verifiable, and what amount of knowledge of HTTP applications is needed in order to do so.

Having said that, when a particular application is in view it is certainly practical to find abstractions appropriate to that application, build models, and devise lists of safety properties one wishes to maintain (absence of deadlock and livelock being only the most trivial of examples); we believe that as web services and other intermediary-driven systems emerge and begin to not only employ more sophisticated and involved features at the HTTP level, but interact among one another in non-trivial and causally related ways, that assessing the correctness of those behaviors in a rigorous and well-understood manner will become increasingly important. Languages like PROMELA and freely available

tools like SPIN make this process accessible to protocol designers and engineers, while the inclusion of the ability to enforce general LTL (Linear Temporal Logic) formulae gives significant expressive power to those who may need it. All of this taken together suggests that it may be prudent to afford a larger place to formal verification of properties in the development of open protocols and of composable Internet services in the future.

6 Conclusion

While one may be tempted to read this paper as a heavy-artillery assault upon the HTTP/1.1 protocol and the HTTP-WG which developed it, that was neither our motivation nor our goal, nor even the emphasis of our result. Our result is the beginning of a rigorous analysis and systematization of one set of properties of the HTTP protocol, namely *safety*, for encoding in a type system which we intend to use to impose stronger disciplines (and therefore safety properties) upon the programming of distributed compositional systems such as edge services.

In pursuing this end, we have shown a number of interesting results. First, our attempts to build models from the RFCs in question highlighted several textual ambiguities (or, arguably, errors): 1. The two interpretations of `100 Continue` in RFC2068 (*proxy-2068-e2e*, *-hbh*, and *-hybrid*), 2. The absence of a backward-compatibility option for RFC2616 proxies (*proxy-2616-fixed*) to mirror the option for RFC2616 servers, and 3. The natural presence of potential deadlock cases for RFC2616 servers choosing not to implement the backward-compatibility option (*server-2616-may*). Second, we have verified that an entirely HTTP/1.1 world (regardless of which revision) is deadlock-free, except in the presence of RFC2616 servers which choose not to implement the backward-compatibility option. Third, our experiments uncovered several classes of agent arrangements in which combinations of HTTP/1.1 agents leading up to an HTTP/1.0 server (or equivalent) agent are prone to deadlock; while this was partially addressed in the RFC2616 revision to HTTP/1.1, certain failure cases remain in environments of mixed RFC2068 and RFC2616 agents. Fourth, from our experiences with these models we were able to construct a set of reduction rules which define classes of arrangements of agents which are equivalent with respect to the safety of their expect/continue behavior. These results, taken together, allow us to exhaustively characterize the set of all possible arrangements of HTTP agents.

In principle, we have illustrated the value of a mechanical verification system to enforce clarification upon a standard document, shown its ability to quickly identifying corner cases for a large number of component-wise and system-wise interactions which may be difficult to discover or argue about by hand, and illustrated how the results of such experiments can be used as groundwork for arguing about the behaviors of more general arrangements of components of arbitrary size, results the likes of which will be of great interest to the designers and implementors of compositional distributed services and applications, and to the tools we expect will emerge to support their work.

Acknowledgements

The authors wish to thank the anonymous reviewers for their thoughtful and helpful comments on this work.

SDG.

References

[1] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0, 1996. RFC1945.

[2] Adam D. Bradley, Azer Bestavros, and Assaf J. Kfoury. Safe composition of web communication protocols for extensible edge services. Technical Report BUCS-TR-2002-017, Boston University Computer Science, 2002.

[3] Adam D. Bradley, Assaf J. Kfoury, and Azer Bestavros. Validating indefinitely large communication networks with finite model checking. Technical Report (work in progress), Boston University Computer Science, 2002.

[4] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1 (obsolete), 1997. RFC2068.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, 1999. RFC2616.

[7] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Also appeared 4th SPIN workshop, Paris, November, 1998.

[8] Gerard J. Holzmann. Designing bug-free protocols with SPIN. *Computer Communications Journal*, pages 97–105, March 1997.

[9] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, May 1997.

[10] Gerard J. Holzmann. From code to models. In *Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)*, 2001.

[11] Gerard J. Holzmann and Margaret H. Smith. A practical method for verifying event-driven software. In *Proc. ICSE99*, pages 597–607, Los Angeles, CA, May 1999.

[12] Gerard J. Holzmann and Margaret H. Smith. Software model checking: Extracting verification models from source code. In *Proc. PSTV/FORTE99 Publ. Kluwer*, pages 481–497, Beijing China, October 1999.

[13] Balachander Krishnamurthy and Martin Arlitt. PROCOW: Protocol compliance on the web. Technical Report HA1630000-990803-05TM, AT&T Labs-Research, August 3 1999.

[14] Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. In *Proceedings of the WWW-8 Conference*, Toronto, May 1999.

[15] Balachander Krishnamurthy and Jennifer Rexford. En passant: Predicting HTTP/1.1 traffic. In *Proceedings of Global Internet Symposium*, December 1999.

[16] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3)(3):219–246, September 1989.

[17] Jeffrey Mogul. Is 100-Continue hop-by-hop?, July 7, 1997. HTTP-WG Mailing List Archive, http://www-old.ics.uci.edu/pub/ietf/http/hypermail/1997q3/.

[18] Jeffrey C. Mogul. Clarifying the fundamentals of HTTP. In *WWW-2002*, Honolulu, HI, May 2002.