Safe Composition of Web Communication Protocols for Extensible Edge Services [*]

BUCS-TR-2002-017

Adam D. Bradley, Azer Bestavros, and Assaf J. Kfoury
artdodge,best,kfoury@cs.bu.edu

Computer Science Department
Boston University
Boston, MA 02215

May 22, 2002

Abstract

As new 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., livelocks) there needs to be a methodology 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 uncovered multi-version interoperability problems, including some which motivated revisions of HTTP/1.1 and some which persist even with the current version of the protocol. One such problem resembles a classic degradation-of-service attack, but can arise between well-meaning peers. We also discuss how the methods we employ can be used to make explicit the requirements for hardening a protocol's implementation against potentially malicious peers, and for verifying an implementation's interoperability with the full range of allowable peer behaviors.

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

[*] 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)

1 Introduction

Motivation and Overview: 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 [KMK99] 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 conformant implementations of HTTP/1.0 (RFC1945) and the now-obsoleted RFC2068 (HTTP/1.1) [Mog97].

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, which is both morally comforting and technically very 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 [Hol97a]. The PROMELA language and the accompanying SPIN verifier [Hol97b], 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 HTTP protocol. In so doing, we identified instances of failure modes under client-proxy-server operating arrangements–some known and some apparently not well-known in the literature. We 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 metadata 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 standalone 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); we can also then build type inference systems which could, for example, mechanically deduce the need for additional agents along a path.

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 reversible set of reduction rules which can also be applied to "stretch" a system to include more agents; (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. 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. Also, we do not address explicitly the process of hardening HTTP agents, as we are still refining our notion of a "bi-directional maximal automaton" which can be harnesses to our proxy models. However, one important implementation hardening result emerged directly from our correctness verifications, and will be discussed.

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.

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 protocol [BLFF96, FGM$^+$97, FGM$^+$99]. 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 something shunned by many in the hacking and software engineering communities, it is worth re-stating some of the things 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 ambiguity and incompleteness of coverage. 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*.

[1] Type systems provide one methodology, out of several ones 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] And, neither do we address the issue of validating implementations thereof.

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 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 real world 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 conforming to our protocol to behave poorly, or to have its resources in some way monopolized? 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 [LT89] 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 [HS99a, HS99b, Hol01] and Java [CDH$^+$00, HP00], is cause for hope.[3]

3 Verifying HTTP: A Case Study

In this section we present our methodology for the safe composition of Web communication protocols in NetBench, by working through a case study that assesses the correctness and interoperability of the various revisions of HTTP as specified in RFCs 1945, 2068, and 2616. 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] If the basic conformance results of [KA99] were any indication, this hope is much-needed.

3.1 A Propos

One of the desired features for the 1.1 revision of the wildly-popular HTTP protocol was the ability for clients to avoid submitting very large documents 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 a bad URL, authentication failure, or temporary server condition) [KMK99]. Conceptually, this mirrors the conditional operators (such as the `If-Modified-Since` header) which allow a response entity to be suppressed if its transmission is unnecessary. This feature is found in the latest revision of the 1.1 specification with the paired mechanisms of *expectation* and *continuation* [FGM$^+$99, §8.2.3, §10.1.1].

While the original specification of the continuation mechanism (via the 100 Continue response header, as governed by RFC2068 [FGM$^+$97, §8.2 and §10.1.1]) was clearly sound with respect to the simple client-server cases, there was ambiguity as to the correct behaviors of intermediary proxies; compelling arguments could be made that the RFC recommended, required, or even alluded that the mechanism be applied either hop-by-hop or end-to-end with respect to a chain of proxies. Under at least one of these interpretations, certain combinations of correctly implemented components in the client-proxy-server chain were prone to deadlock. This problem was itself dealt with in the revision of the spec (RFC2616 [FGM$^+$99]) by the introduction of the `Expect` mechanism and the clarification of the semantics of `100 Continue` with respect to proxies; however, the fact that many existing 1.1 implementations conform to earlier versions of the spec also demanded additional heuristics be included in the revision to attempt to ensure correct interoperation with those implementations. This quagmire of versions of the spec, 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 [Hol97b] from Bell Labs to construct and verify models of the expect/continue behavior of clients, proxies, and servers conforming to RFC1945 (HTTP/1.0) [BLFF96], multiple interpretations of RFC2068 (obsolete HTTP/1.1) [FGM$^+$97], and RFC2616 (HTTP/1.1) [FGM$^+$99].[4]

3.2 A Brief Introduction to Promela and SPIN

Promela, the PROcess (or PROtocol) MEta LAnguage, is the input language to the SPIN verifier [Hol97b]. Promela is a non-deterministic guarded command language, which means 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. Coding up deterministic state machines in Promela is a trivial exercise to anyone familiar with imperative programming and the syntax of C, and blocks containing sets of predicated but nondeterministically chosen paths are similarly trivial to code.

Promela provides us with 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. Common practice for the message queue is to send a message with a "type" (a member of the enumerated set `mtype`) and an accompanying arbitrary set of values.

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 the equivalent of a depth-first search of execution paths of the whole program, searching for cases which violate some set of constraints. The easiest property to test for is deadlock[5]; if the system can reach a state in which one or more processes have not terminated and no process has a runnable instruction[6], then the system is deadlock-prone, and the execution path leading to an instance of deadlock is output to the user. Since depth-first search can easily lead to extremely long exemplars, the system can then continue the search while bounding the depth, looking for shorter execution paths which produce errors, until it has found the shortest one.

[4] None of the current errata for RFC2616 listed at `http://purl.org/NET/http-errata` pertain to the expect/continue mechanism.

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

[6] That is, all processes are "asleep" waiting for predicates to change or events to occur

Using Promela, an execution path can be presented graphically to the user as a message sequence chart, showing ordering constraints of send and receive events in each of the participating agents; Figure 1 is an example of a partial message sequence chart. If no error cases are found, the system reports success to the user.

3.3 A Promela Model of HTTP Expect/Continue

The key to building useful and analyzable models is for the model to abstract away enough details to make it 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 (`mtypes`):

request - corresponds to the "Request" grammar in [FGM$^+$99, §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[7] value.
- **hasentity** - Boolean indicator of whether a `message-body` will follow the request. This is an abstraction of the rules for inclusion found in [FGM$^+$99, §4.3, ¶5].
- **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 [FGM$^+$99, §6] up to and including the `CRLF`, but excluding the `[ message-body ]`. This message c-arries a parameter structure with the following fields:

- **version** - As above.
- **hasentity** - As above; abstraction for the rules in [FGM$^+$99, §4.3, ¶6].
- **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 [FGM$^+$99, §10.1, §10.1.1].[8]

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 the connection down.

This data is sufficient to control all of the behaviors surrounding Continuation described in [FGM$^+$97, §8.2, §10.1, §10.1.1] and [FGM$^+$99, §8.2.3, §10.1, §10.1.1, §14.20]. We do not currently model the backoff-and-retry mechanism discussed in [FGM$^+$99, §8.2.4].

[7] Useful for testing whether the model is hardened against arbitrary behaviors of "future" protocol revisions

[8] To correctly implement [FGM$^+$99, §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 Footnote 20; hence, we omit this from our models.

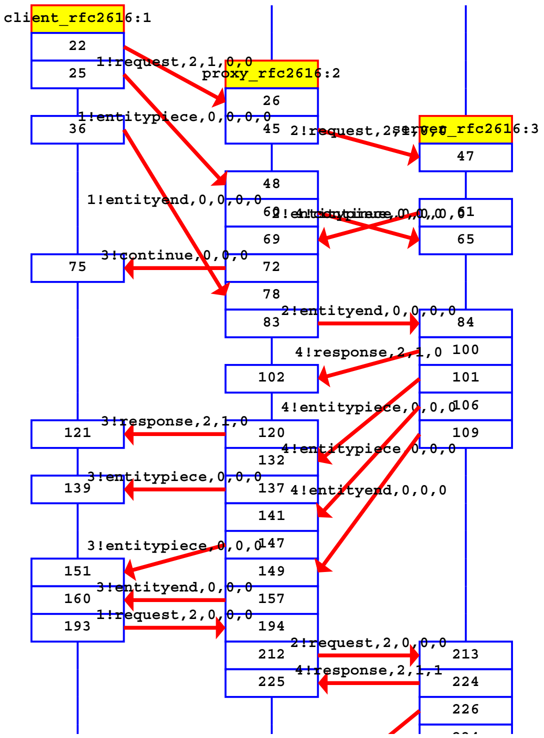

Figure 1: A sample message sequence chart

3.4 Implementations of Agents

For each role defined by HTTP (client, proxy, server) and for each revision of the specification (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 is actually a simplification which allows us to exhaust the space of transaction sequences without having to spawn multiple agents; it also naturally emulates the "upstream version cache" features appearing in HTTP/1.1 specifications. The presence of one-per-connection HTTP/1.0 transactions in the "real world" does not bear in any meaningful way upon our results.[9]

Table 1 lists all of the models, the RFC 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 surprisingly readable once one understands how the non-deterministic selection (`if ... fi`) and looping (`do ... od`) constructs work; the main body of the *server-2068* implementation is presented in appendix A as an example.

Disambiguating the RFCs: For the HTTP/1.0 proxy, we model it essentially 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 the response entity in "write-through" style, but that behavior would be causually indistinguishable to its peers from the whole-response method we employ.

[9] However, the presence of virtual servers or proxies made up of multiple agents implementing different versions of the spec [KA99] could interact poorly with the upstream version cache, a problem that could be exacerbated by the absense of end-to-end persistent connections. We have not yet modeled such situations, although our modeling results do suggest under which conditions these problems may arise.

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 [FGM+99, §8.2.3 ¶8] is "MUST after timeout"
server-2616-may	RFC2616 (1.1)	Server	150	implements [FGM+99, §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 page 12)

Table 1: HTTP Agent Models

With regard to the behavior of proxies, RFC2068 is ambiguous and allows several 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, wherein 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; tells the client to continue, reads its `message-body`, then forwards the request upstream and waits for a "Continue" message.
- *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, and may not be reflective of the actual behavior of any actual RFC2068 implementation.

We also have made *server-2616* always employ the interoperability clause in [FGM+99, §8.2.3 ¶8] when the system enters a timeout[10] state. 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.[11]

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

[10] In Promela, the "timeout" guard can be thought of as a "global else"; it becomes executable only if no other instructions throughout the model (except perhaps other timeouts) are executable.

[11] 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 now convinced themselves that a `Continue` message will be produced by this server since it self-identifies as HTTP/1.1.

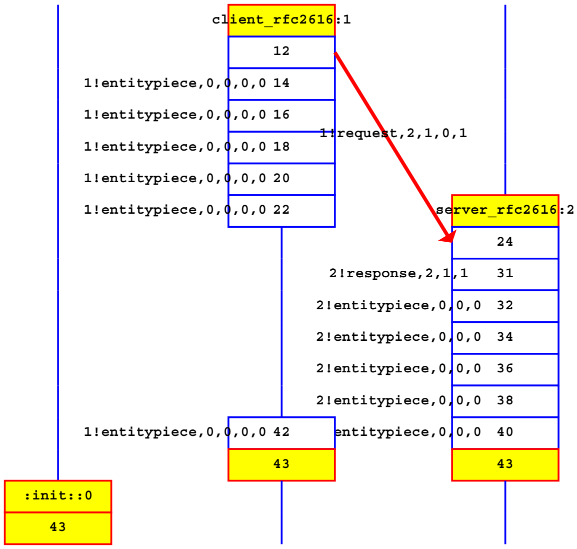

Figure 2: MSC for Buffer Write Deadlock

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 not a tractible problem.

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[12]), S is the number of server models (three), and N is a heuristic bound on the length of the proxy chain. Again, this quickly becomes intractible as N grows, so other means are needed to reason about lounger chains.

3.6 Correctness Results

We were rather surprised when, in 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 arrives as follows: Imagine that the client sends a request, but does not set the `Expect: 100-continue` header. The server receives the request, and promptly decides to reject the request, and 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.[13] This situation is illustrated by the MSC in Figure 2. The finite buffers in each direction, each being simultaneously written to, gives rise to the problem: since both agents are writing and not reading, both buffers become full, and if the agents are implemented using blocking write operations[14],

[12] Neither the *proxy-2068-e2e* nor the *-hbh* model are considered "principal" models and are not considered in the initial interoperability exeriments. This is because the *-hybrid* model is a true superset of the two other *proxy-2068* implementations. Said another way, any legal execution of an *-e2e* or *-hbh* model is equivalent in all regards to some legal execution of t he *-hybrid* model. Thus, if a system validates with the *-hybrid* node acting in any place in the arrangement, we know that an *-e2e* and *-hbh* node would also validate, so we choose to forego inclusion of the two special case models in our initial tests, reducing P to 3. However, should an arrangement including the *-hybrid* model fail to validate, we can use the other two more specialized models to study how and why this happens. It is also worth noting that the time required to verify an arrangement including a *-hybrid* agent is significantly greater than that to verify one without. This is not surprising; its finite state model is essentially a concatenation of the other two, with some minimal sharing of the outermost loop code. While this is not a significant obstacle to $N = 1$ arrangements, $N = 2$ or higher arrangements including a *-hybrid* proxy agent can be computationally very expensive to verify, further motivating alternative means to argue about larger arrangements.

[13] 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.

[14] This is common in threaded and process server architectures

	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: Localizing Interoperability Problems for $P = 1$

they will deadlock. 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.

Technically speaking, this deadlock is an engineering concern; however, it can now arise in totally benevolent environments because of the more involved communication model of HTTP/1.1 in which both peers may be sending a message simultaneously. 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. This is a happy coincidence of good closed-system engineering and hardening of the system to interact with the open-system world.

To remove this deadlock from all of our models, we changed the macro which produced arbitrarily long entities (both for requests and responses) to produce variable but short ones (one or two **entitypiece** messages followed by an **entityend**), and we enlarged the buffers in each direction from 4 to 6 messages. This allowed the entire modeled sequence of messages constituting any client's or server's part in the most involved of transactions to fit comfortably in the buffers.[15]

Having removed the buffering deadlock from our models, we found that all our RFC2616-only arrangements were verified by SPIN without further problems. The verification of the *client-2616→server-2616*, *client-2616→proxy-2616→server-2616*, and *client-2616→proxy-2616→proxy-2616→server-2616* arrangements required less than 30 seconds of CPU time[16]; more time was required to compile the verifiers for each arrangement than to actually run them.

3.7 Interoperability Results

SPIN required only a few seconds to test all nine client-server cases, and was able to do so by verifying a single model which explicitly included all nine possible pairings. As none of these cases includes any possible deadlock case, 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, because there are in fact arrangements which are deadlock-prone; testing a single model prevents us from differentiating which cases are and are not correct;

[15] Our first instinct was to integrate back-logging non-blocking write operation into our models; however, constraints of the SPIN verification tool make it difficult to write such constructs in ways that are friendly to look at. Instead, we removed this deadlock by doing away with the possibility of a "Hold and Wait" on the two buffers during the course of processing a single transaction.

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

therefore, we began by examining large subspaces of the 27 possible arrangements and then "drilled down" into those subspaces which exhibited failure modes.

The final results of this process are briefly summarized by the list of experiments[17] in Table 2 and the interoperability chart in Table 3, where each result in the latter 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.

	→*server* models		
client→*proxy*	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

Classic 1.1/1.1/1.0 Deadlock: The first set of deadlocks, represented in experiments 2.5 and 2.6, are reflected by the example MSC shown in Figure 3. These cases were identified in 1997 on the HTTP-WG mailing list [Mog97], and we refer to them as the "classic" continuation deadlock, in 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).[18]

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 shortest example MSC is virtually identical to Figure 3. 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 headers 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) [FGM^{+}99, §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.

[17] 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).

[18] 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.

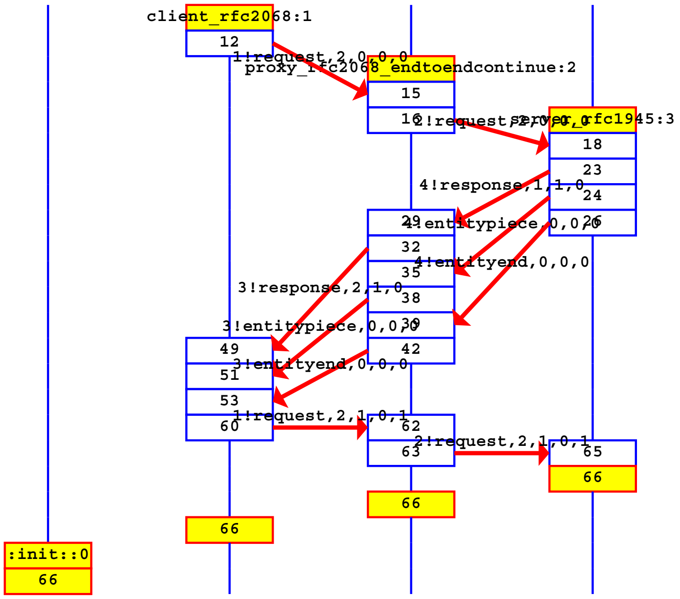

Figure 3: Example MSC for 1.1 Client/2068 Proxy/1.0 Server Deadlock

This deadlock can be resolved by altering the spec and borrowing an idea from a compatibility rule for origin servers:

> [F]or compatibility with RFC2068, a server MAY[19] send a 100 (Continue) status in response to an HTTP/1.1 PUT or POST request that does not include an Expect request-header field with the “100-continue” expectation. This exception [...] applies only to HTTP/1.1 requests, and not to requests with any other HTTP-version value. [FGM$^+$99, §8.2.3 ¶8]

By allowing (requiring?) proxies to initiate their own `100 Continue` or `417 Expectation Failed` 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 4. 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 [FGM$^+$99, §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 autonymously) 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).[20]

[19] We have already addressed in footnote 11 why we treat this as a MUST behavior in our models.

[20] 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 [FGM$^+$99, §10.1] which describes proxy forwarding behavior for 1xx messages says that “There are no required headers for this

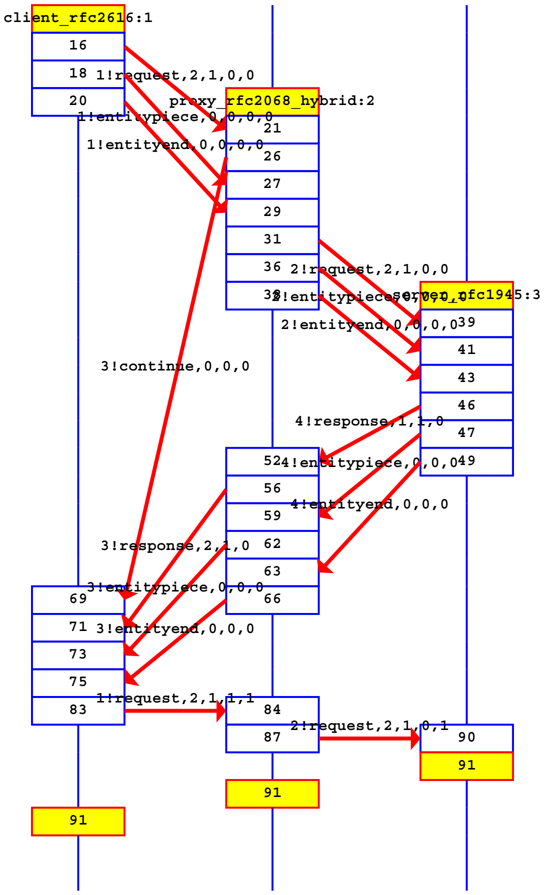

Figure 4: Example MSC for Hybrid Proxy Deadlock

Longer Chains: A summary of experiments for client-proxy-proxy-server arrangements is presented in Table 4[21], with the safety matrix of the 81 arrangements (excluding *-hbh* and *-e2e* agents) presented in Table 5 using the same format as for previous results.

We naturally expect that many of the deadlock cases which arise in these experiments will mirror cases found in table 2, and this is indeed the case; we have marked such cases in table 5 with examples of corresponding experiments. These relationships, the derivation of which is beyond the scope of this paper, helped us in deducing the set of reduction rules which appears below.

We notice immediately that many deadlocks arise in parallel 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 the *proxy-2616-fixed* model; we repeated all of the $N = 2$ failure cases

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).

[21] Wherever the word "each" appears in Table 4, it reflects a set of verification runs in which each of the models is individually tested in that role, and every such individual configuration produces the same result (clean or deadlock); where the "all" wildcard and a deadlock result indicated the presence of *at least one* deadlock arrangement within the wildcard's scope, the "each" wildcard and a deadlock result indicates that *all* arrangements within the wildcard's scope are deadlock-prone.

	Client	Proxy	Proxy	Server	Result
4.1	all	1945	all	all	clean
4.2	1945	all	1945	all	clean
4.3	each of 2068,2616	2068-hybrid	1945	each	deadlock
4.4	2068	2068-e2e	1945	each	deadlock
4.5	2068,2616	2068-hbh	1945	all	clean
4.6	2068	2616	1945	each	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	each	2068	2616	1945	deadlock
4.11	1945,2616	2616	2616	1945	clean
4.12	2068	2616	2616	1945	deadlock
4.13	each of 2068,2616	each of 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 $P = 2$

which included a *proxy-2616* agent by replacing all such agents with the *-fixed* variant, and found then that every one of those cases validates.

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 protocol (e.g., HTTP) compositions. One of the hurdles that face many tools such as SPIN is the state space explosion, which raises concerns as to their scalability for non-toy problems. In this section, we show that state space reduction rules based on domain-specific knowledge could be devised to 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, it would seem reasonable to infer that, for these models, there are deadlock-prone *patterns* which longer chains could be checked for to determine whether they will deadlock or not. This leads us to postulate a set of reduction rules, according to which we can organize and partition the (potentially enormous) search space of arrangements into equivalent classes of success patterns and failure patterns, which we believe can be used to describe the set of deadlock-prone chains of arbitrary length.

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, theoretical sense) of arrangements which fall into either category. Toward that end, we here describe two sets of relations: First are a set of 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.

	→server models		
client→proxy→proxy	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

While we are exploring techniques for deducing these equivalence relationships mechanically rather than manally, 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 its downstream (toward the client) agents. However, it is distinguishable from a client-1945 to upstream (toward the server) agents in that it might (sometimes does, sometimes does not) pass through the “expect-100” token. Since this token will not be passed on by anything but a next-hop *proxy-1945* (which we show can be reduced away by Reduction 2), will be ignored (as an unknown extension) and not be forwarded (since it is marked by the `Connection` header as being hop-by-hop) by *server-2068* and *proxy-2068* agents (respectively), and MUST be ignored by *proxy-2616* and *server-2616* agents when coming from an HTTP/1.0 client per [FGM^{+}99, §14.10 ¶10], it is safe to treat *proxy-1945* as a *client-1945* with respect to its upstream agents. Thus, the sequence *x→proxy-1945→y* is verified if the two chains *x→server-1945* and *client-1945→y* are verified.

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.

3. When a *proxy-2616* immediately follows a *client-2616*, it is essentially equivalent to an arrangement in which the *proxy-2616* is absent; this hold true because the *proxy-2616* in all regards with respect to expectation/continuation

acts as a repeater for the client, leaving all request parameters intact (including the presence/absence of `Expect: 100-continue`) and neither filtering any messages out nor initiating any on its own in either direction. This reduction can then be re-applied inductively; thus, any sequence of the form *client-2616*→(*proxy-2616*→)*x is equivalent to *client-2616*→*x*.

4. The same can be said of *proxy-2616* when it immediately precedes *server-2616*; thus, *x*→(*proxy-2616*→)*server-2616 is equivalent to *x*→*server-2616*.

5. Also for similar reasons, a series of *proxy-2616* agents anywhere in the arrangement reduces to a single *proxy-2616*.

6. A series of *proxy-2068-e2e* agents reduces to a single *proxy-2068-e2e* for the same reasons as 5 (the pass-through, non-initiating behavior).

7. For the reasons similar to 3 and 4, a *proxy-2068-e2e* node can also be collapsed into an immediately adjoining *server-2068* node; this does not work for adjoining *client-2068* nodes, however, because by doing so we could potentially change the response version number seen by that client from HTTP/1.1 to HTTP/1.0, which has an effect upon the behavior of RFC2068 clients.

8. Our model of a *proxy-2068-hbh* agent is indistinguishable from a *server-2068* to downstream agents (in the it collects whole requests of its own accord, then provides full responses), and indistinguishable from a *client-2068* to upstream agents (in that it sends the parts of a request entirely under its own volition); thus, by the same argument as 1, any *x*→*proxy-2068-hbh*→*y* arrangement verifies if and only if *x*→*server-2068* and *client-2068*→*y* verify.

9. By applying 8 and the same argument as in 2, chains of *-hbh* agents can be reduced to a single agent.

10. Unfortunately, neither 6 nor 8 can be applied directly to *proxy-2068-hybrid*, as is proven by experiment 4.14; if series of *-hybrid* nodes were collapsible (as in 6 then that experiment would verify because (per experiment 2.1) *client-1945*→*proxy-2068-hybrid*→*server-1945* verifies, and if *-hybrid* were generically replaceable with *client-2068* and *server-2068* (as in 8) then the experiment would verify because (again per 2.1) *client-1945*→*proxy-2068-hybrid*→*server-2068* verifies.

11. Notwithstanding 10, taking a more careful look at the interactions between *client-1945* and an immediately upstream *proxy-2068-hybrid* we find that it is true that to upstream nodes that particular pair is indistinguishable from a *client-2068*; the request version will always be HTTP/1.1, there will never be an `expect100` flag, and they may or may not wait for a `Continue` message (depending on whether *proxy-2068-hybrid* assumes its *-e2e* or *-hbh* persona). As a result, we are able to reduce *client-1945*→*proxy-2068-hybrid*→*x* to *client-2068*→*x*.

12. Also notwithstanding 10, all series of *proxy-2068-hybrid* agents of length greater than 2 are equivalent to one of length 2. Consider that each *-hybrid* agent must choose, at the beginning of each transaction, whether to act in its *-e2e* or *-hbh* persona. First and foremost, we collapse all sequences of *-e2e* agents per 6, and collapse all sequences of *-hbh* by 9, leaving us with a sequence of alternating *-e2e* and *-hbh* agents. Applying *en:hbhremoval* and experiment 2.3, we can collapse everything between the first and last *-hbh* nodes, as each *hbh-e2e-hbh* subsequence validates internally. This leaves us with five possible chains: *hbh*, *e2e*, *e2e-hbh*, *hbh-e2e*, and *e2e-hbh-e2e*. By applying reduction 7, we find that this set is further reducible to *hbh*, *e2e* and *hbh-e2e*, all three of which are instantiations of the behaviors of a sequence of two *proxy-2068-hybrid* agents.

13. Where an arrangement includes a *client-1945*, a number of reductions apply because of the simplicity of its communication model. If the first upstream agent is a *proxy-2616*, for example, then that proxy's behavior will be a subset of the behavior of a *client-2616*; more precisely, it will never set the `expect100` flag, and will never wait to transmit the request entity. This implies that if *client-2616* would verify in place of *client1945*→*proxy-2616*, then the latter would verify as well; however, if the first fails, the second would not necessarily (since the first is capable of behaviors the second is not). A similar argument applies when the first upstream agent is *proxy-2068-hybrid*, except that the proxy may behave according to the full range of permissible *client-2068* behaviors, in that it may or may not wait for a `Continue` message before sending an entity depending on whether it selects its *-e2e* or *-hbh* persona; thus, *client-1945*→*proxy-2068-hybrid*→*x* is reducible to *client-2068*→*x*.

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 those 19 are reducible to clean client-server cases ($N = 0$). Those five exceptions all involve the peculiarities of interactions between RFC2068 and RFC2616-conforming HTTP/1.1 agents; further valid reductions do exist to explain some of these cases, but they are even more involved and less intuitively clear than the ones given above.

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 Footnote 11) and are interacting with an HTTP/1.1 proxy immediately downstream.

1. Any arrangement in which a *proxy-2616* must interact with an RFC2068 agent immediately downstream (whether client or proxy) and a *server-1945* (or equivalent) agent immediately upstream will be deadlock-prone, as discussed on 11 in connection with experiment 2.8. This rule corresponds with experiments 2.8, 4.6 and 4.10.
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.

These failure classes, combined with the reduction rules above, are sufficient to explain all of the deadlock cases for the $N = 1$ arrangements; all but one actually match one of the two patterns directly, the one exception requiring the application of reduction 11 to yield a match. Similarly, all but two $N = 2$ arrangement deadlocks are proven directly by matching these patterns; the other two match after applying reductions 11 and 5.

While we have not yet rigorously proven this result, we believe that these patterns and reductions exhaustively identify the deadlock-prone cases for our models, and from them we should in principle be able to build a formal language whose members describe all deadlock-prone arrangements, and therefore also to construct a language which describes its compliment, that is, the set of all deadlock-free arrangements.

5 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 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.

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 absense 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.

Fifth, we were able to argue for two distinct patterns of agents which are inherently prone to a continuation deadlock, one centering around *proxy-2068* and one around *proxy-2616*.

In principle, we have illustrated the value of a mehanical 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 deduce or argue about by hand, and illustrated how the results of such exeriments 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.

SDG.

A Sample Promela: server-2068

Some of the more opaque details have been removed from the code. Within each `if ... fi` and `do ... od` block, a `::` indicates one possible execution (a selection); each is followed by an expression (the "guard") and zero or more statements; among those selections whose guards evaluate to "true", one will be executed, and if none are true, the construct blocks.

```
inline send_general_2068_response()
{
  if /* keepalive or not?  our prerogative */
  :: true -> respdata.close = true; /* force close */
  :: true -> respdata.close = reqdata.close; /* defer to client */
  fi;
  if /* might have entity, might not - imagine 304/412 */
  :: true -> respdata.hasentity = true;
  :: true -> respdata.hasentity = false;
  fi;
  downstream!response(respdata);
  if
  :: respdata.hasentity -> SEND_ENTITY(downstream);
  :: else ->
  fi;
}

/* RFC2068 HTTP/1.1 implementation */
proctype server_rfc2068(chan upstream, downstream) {
  xr upstream;
  xs downstream;

  req reqdata;
  resp respdata;
  respdata.version = RESP_HTTP_11;
  do
  :: upstream?request,reqdata ->
    if
    :: reqdata.hasentity -> /* See 8.2, para 5, 6 */
      if /* do we want it? */
      :: true -> /* random: we want it */
        if /* honor para 5: MUST NOT */
        :: reqdata.version > REQ_HTTP_10 -> downstream!continue(0);
        :: else ->
        fi;
        do /* wait for the request body */
        :: upstream?entitypiece,reqsink -> skip;
        :: upstream?entityend,reqsink -> break;
        :: upstream?eof,reqsink -> goto shutdown; /* client bailed */
        :: BAD_CLIENT_EVENT(upstream,response);
        :: BAD_CLIENT_EVENT(upstream,continue);
        :: BAD_CLIENT_EVENT(upstream,request);
        od /* request entity completely read */
      :: true -> /* random: we don't want it */
        send_general_2068_response();
        if
        :: respdata.close -> /* per para 5 */
          goto shutdown;
        ::  else -> /* keepalive: read off the rest */
          do
          :: upstream?entitypiece,reqsink -> skip;
          :: upstream?entityend,reqsink -> break;
          :: upstream?eof,reqsink -> goto shutdown; /* client bailed */
          :: BAD_CLIENT_EVENT(upstream,continue);
          :: BAD_CLIENT_EVENT(upstream,response);
          :: BAD_CLIENT_EVENT(upstream,request);
          od;
          goto done;
        fi;
      fi;
    :: else -> /* no entity */
      skip;
    fi;
    send_general_2068_response();
    if
    :: respdata.close -> goto shutdown;
    :: else ->
    fi;
  done:
    skip;
  :: upstream?eof,reqsink -> break; /* client bailed - nothing left to do */
  :: BAD_CLIENT_EVENT(upstream,entitypiece);
  :: BAD_CLIENT_EVENT(upstream,entityend);
  :: BAD_CLIENT_EVENT(upstream,continue);
  :: BAD_CLIENT_EVENT(upstream,response);
  od;
shutdown:
  downstream!eof(0);
}
```

References

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk, *Hypertext transfer protocol – HTTP/1.0*, 1996, RFC1945.

[CDH$^+$00] 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*, Proceedings of the 22nd International Conference on Software Engineering, June 2000.

[FGM$^+$97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, *Hypertext transfer protocol – HTTP/1.1 (obsolete)*, 1997, RFC2068.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext transfer protocol – HTTP/1.1*, 1999, RFC2616.

[Hol97a] Gerard J. Holzmann, *Designing bug-free protocols with SPIN*, Computer Communications Journal (1997), 97–105.

[Hol97b] Gerard J. Holzmann, *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), no. 5, 1–17.

[Hol01] Gerard J. Holzmann, *From code to models*, Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01), 2001.

[HP00] K. Havelund and T. Pressburger, *Model checking Java programs using Java PathFinder*, Int. Journal on Software Tools for Technology Transfer **2** (2000), no. 4, 366–381, Also appeared 4th SPIN workshop, Paris, November, 1998.

[HS99a] Gerard J. Holzmann and Margaret H. Smith, *A practical method for verifying event-driven software*, Proc. ICSE99 (Los Angeles, CA), May 1999, pp. 597–607.

[HS99b] Gerard J. Holzmann and Margaret H. Smith, *Software model checking: Extracting verification models from source code*, Proc. PSTV/FORTE99 Publ. Kluwer (Beijing China), October 1999, pp. 481–497.

[KA99] Balachander Krishnamurthy and Martin Arlitt, *PRO-COW: Protocol compliance on the web*, Tech. Report HA1630000-990803-05TM, AT&T Labs-Research, August 3 1999.

[KMK99] Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol, *Key differences between HTTP/1.0 and HTTP/1.1*, Proceedings of the WWW-8 Conference (Toronto), May 1999.

[LT89] Nancy Lynch and Mark Tuttle, *An introduction to input/output automata*, CWI-Quarterly **2(3)** (1989), no. 3, 219–246.

[Mog97] Jeffrey Mogul, *Is 100-Continue hop-by-hop?*, July 1997, HTTP-WG Mailing List Archive, http://www-old.ics.uci.edu/pub/ietf/http/hypermail/1997q3/.