Systematic Verification of Safety Properties of Arbitrary Network Protocol Compositions Using CHAIN [*]

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

BUCS-TR-2003-012

Abstract

Formal correctness of complex multi-party network protocols can be difficult to verify. While models of specific fixed compositions of agents can be checked against design constraints, protocols which lend themselves to arbitrarily many compositions of agents–such as the chaining of proxies or the peering of routers–are more difficult to verify because they represent potentially infinite state spaces and may exhibit emergent behaviors which may not materialize under particular fixed compositions. We address this challenge by developing an algebraic approach that enables us to reduce arbitrary compositions of network agents into a behaviorally-equivalent (with respect to some correctness property) compact, canonical representation, which is amenable to mechanical verification. Our approach consists of an algebra and a set of property-preserving rewrite rules for the Canonical Homomorphic Abstraction of Infinite Network protocol compositions (CHAIN). Using CHAIN, an expression over our algebra (*i.e.*, a set of configurations of network protocol agents) can be reduced to another behaviorally-equivalent expression (*i.e.*, a smaller set of configurations). Repeated applications of such rewrite rules produces a canonical expression which can be checked mechanically. We demonstrate our approach by characterizing deadlock-prone configurations of HTTP agents, as well as establishing useful properties of an overlay protocol for scheduling MPEG frames, and of a protocol for Web intra-cache consistency.

1 Introduction

Increasingly, the Internet is being used as a ubiquitous infrastructure supporting a multitude of distributed applications and services. Unfortunately, the introduction (deployment and versioning) of Internet services is laden with uncertainties that arise from our inability to formally establish the safety of such services—namely, that such services will not interfere with existing services, or even with older versions of the same service.[1] In light of the critical role that Internet services play in today's economy and society, it is incumbent on the networking community to develop sound formalisms for (and promote the use of such formalisms in) supporting/enforcing software engineering practices for distributed applications deployed over the Internet. Indeed, in a recent NSF PI meeting of over 250 network researchers, improving the trustworthiness of Internet applications [27] and the development of formalisms to scale our understanding of networked systems [26] were deemed to be (two of) the most pressing challenges facing the networking community for the next few years.

Current efforts to formally establish (or assess) the "trustworthiness" of the Internet have focused on proving desirable properties[2] of an *individual* protocol or service agent [10], with very few efforts focusing on properties that emerge from the *composition* of such protocols and services (see [14] for one recent example). Properties that emerge from the composition of network protocols are much harder to reason about—not to mention check mechanically—due to the arbitrary nature of such compositions, which is made even more challenging when we consider issues of scalability and variant protocol implementations. We motivate this with a concrete example drawn from the networking community's experience with the HTTP protocol.

HTTP Protocol Compositions: 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 [19] 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) [24].

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

[*]This research was supported in part by NSF (awards ANI-9986397, ANI-0095988, CCR-9988529, ITR-0113193, and ANI-020205294) and U.S. Department of Education (GAANN Fellowship)

[1]The Web community's experiences in relationship to the evolution of HTTP–as well as other findings we present in this paper—underscore this state of affairs.

[2]These range from safety properties (*e.g.*, absence of deadlocks, buffer overruns) to security properties (*e.g.*, authentication, privacy) to compliance properties (*e.g.*, TCP friendliness).

a difficult problem for engineers at all layers of the stack. These problems arise in such settings even without the complexities of multi-version interoperability placed upon HTTP/1.1 and its revisions.

In this paper we show how to use an algebraic approach coupled with a model-checking engine to *systematically*[3] discover unsafe behaviors of arbitrary compositions of HTTP agents, including fully characterizing deadlock conditions we previously uncovered [5]. We also show how this same approach generalizes to the verification of other network protocols, including an overlay network protocol for scheduling of MPEG frames and a Web intra-cache consistency protocol.

Paper Contributions and Overview: This paper proposes a systematic approach to the verification of safety properties in arbitrary compositions of network protocols using CHAIN–an algebra and associated rewrite rules for the *C*anonical *H*omomorphic *A*bstraction of *I*nfinite *N*etwork protocol compositions. We instantiate our approach for a number of network protocols—showing how it enables us to identify safety violations of *arbitrarily large compositions* of these protocols mechanically, using readily-available model checking technologies.

Using CHAIN, the composition (*i.e.*, chaining) of network protocol agents is represented symbolically using strings (or chains), whereby each chain represents a possible "path" through (or arrangement of) the subset of agents that comprise the composition. Arbitrary compositions of a set of network protocols[4] are likely to result in a large (possibly infinite) number of such paths. Thus, such arbitrary compositions can be algebraically represented using a possibly infinite set of chains (each for a given path through the set of agents). To prove that a particular property holds for such arbitrary compositions requires us to verify that it holds for *every* chain in that set. To make this process practical, we develop a set of *rewriting rules* over the CHAIN algebra. These rewriting rules preserve the property under consideration. By repeatedly applying these rewriting rules, we effectively *reduce* the set of chains representing arbitrary compositions to a canonical (minimal, much smaller) set of chains—an abstraction of the original set. This abstraction is a homomorphic image of the original set in the sense that if a property holds for the abstraction, then it provably holds for *any* of the possibly infinite compositions (or arrangements) of the network protocols. Where the homomorphic image is of finite size, the complete abstraction can then be verified mechanically using off-the-shelf model-checking tools.

The remainder of this paper is organized as follows. As a motivation for (and a case study for the application of) our methodology, we begin in Section 2 by presenting the HTTP request continuation mechanism, a feature of the HTTP/1.1 protocol. We follow that in Section 3 with a presentation of the underpinnings of our CHAIN algebra and reductions. In Section 4 we bring the formalisms in CHAIN to bear on three examples of protocol compositions. First, we use it to characterize possible safety violations (deadlock scenarios) of HTTP/1.1 protocol compositions. We do so by translating previously established "equivalence" relationships into algebraic rewrite rules, thus creating a finite homomorphic image of the infinite set of HTTP compositions, allowing us to exhaustively identify the infinite sets of deadlock-prone and deadlock-safe compositions. Next, we illustrate the application of CHAIN to two additional network protocol composition problems: an MPEG packet routing protocol for overlay networks, and a protocol for ensuring web intra-cache consistency. We conclude the paper in Section 6 with a summary and a brief discussion of future directions of this work.

[3]"Systematically" does not necessarily mean "automatically". Rather, it means "methodically", or subject to a prescribed process.

[4]When we refer to "composition of network protocols", we are referring to the composition of multiple agents speaking a particular protocol (*i.e.*, composition of a protocol with itself), not the layering of distinct protocols within a single agent or communication channel.

2 HTTP Request Continuation

In HTTP/1.0, all transactions had a very simple and stateless communication model: (1) A client would send a whole request, *i.e.*, a request line, a set of headers, and an optional request entity; (2) The server, after receiving the whole request, would respond with a complete request, *i.e.*, a status line, a set of headers, and an optional response entity.

One of the desired features for HTTP/1.1 was the ability for clients to avoid transmitting very large entities with their requests when the transaction will fail independent of the content of the document (*e.g.*, an authentication failure or a temporary server condition) [19]. Conceptually, this mirrors conditional operations (such as the `If-Modified-Since` header) which allow a response entity to be suppressed if its transmission is unnecessary. The original HTTP/1.1 specification, RFC2068, supports this capability by allowing clients to pause before sending request entities; the server may send an error code immediately, informing the client that the request has already failed and the request entity should not be sent, or may send a `100 Continue` response, which tells the client to send the request entity (although it does not guarantee that the final response will not still be an error condition).

While the original specification of this mechanism (the `100 Continue` response header [12, §8.2 and §10.1.1]) was clearly sound with respect to simple client-server cases, it was ambiguous as to the correct behavior of proxies; compelling arguments were made that the RFC's language suggested both hop-by-hop and end-to-end interpretations of the feature. It was realized that, under at least one of these interpretations, certain combinations of correctly implemented components in the client-proxy-server chain were prone to deadlock [24]; an attempt at addressing this problem was made in the next public revision (RFC2616) with the introduction of the `Expect` mechanism [13, §8.2.3] and the clarification of the semantics of `100 Continue` [13, §10.1.1] with respect to proxies. Given that many existing implementations conformed to the various interpretations of RFC2068, it was decided that RFC2616 should also include a number of heuristics to facilitate graceful interoperation with those implementations. The resulting quagmire of special-case interoperability rules and the set of possible combinations of revisions in the various roles makes it difficult to say anything with certainty about the correctness and full interoperability of the specification; while it seemed *reasonably* (and even *empirically*) to be correct, it was not *provably* so.

2.1 Model Checking of HTTP Agents

In previous work [5], we presented a set of such models for HTTP clients, proxies, and servers, and discussed experiments with particular configurations. Any single combination of a client, some proxies, and a server (hereafter an *arrangement*) can be examined using a finite-state modeling tool like SPIN [16] which instantiates and joins the models with message channels and determines whether any possible execution of *that* arrangement can lead to an undesirable state (*e.g.*, deadlock, livelock, assertion violation).

To prove the *correctness* of HTTP/1.1 (RFC2616), we need for all client-server and client-proxies-server arrangements of RFC2616 agents to be verified mechanically. The client-server case is straightforward because it consists of a single arrangement. However, in order to say something concrete regarding situations in which there may be arbitrarily many proxies between the client and the server, such an approach is clearly inappropriate, as it would demand verification of an infinite number of arrangements using the model checker (one for each possible length of the proxy chain). Convincing ourselves of the *interoperability* property is a similar process; for a client-proxy-server architecture like HTTP, a brute-force approach requires us to verify as many as

$$|\mathcal{C}| \times |\mathcal{S}| \times \left(\sum_{i=0}^{\infty} |\mathcal{P}|^i \right)$$

arrangements, where $\mathcal{C}$ is the set of client models, $\mathcal{P}$ is the set of proxy models, and $\mathcal{S}$ is the set of server models. Using this brute-force approach, not only would a "complete" proof require verifying an infinite number of arrangements, but even a "partial" proof (all cases up to N proxies) requires verifying a number of arrangements exponential in N.

We have previously established that particular arrangements are provably equivalent (in terms of their cause-effect behaviors) to other arrangements [5]. With a sufficient set of such "behavioral equivalence" relationships, one could potentially reduce an arbitrarily large set of arrangements to a much smaller (preferably finite) set of behaviorally representative arrangements, which could then each be verified. To be useful, the discovery and application of such reductions must follow a systematic approach. Much work has already been done in several communities on discovery of such relationships (*e.g.*, [7, 2, 22]); in the remainder of this paper, we presume such results and develop a *reduction* strategy based upon an algebraic term rewriting technique that is immediately applicable to a wide range of network application structures. [5]

It is important to note that a number of techniques have been developed in the literature for checking models with infinite state spaces (*e.g.*, [17, 9, 21, 8, 20]). These techniques tend (in their current state) to be fairly opaque in the sense that they often cannot connect their abstraction of the infinite state space with intuitively useful declarations about the behavior of particular protocol agents. [6] For designing and testing distributed protocols, we believe that the ability to do so is crucially important not only for purposes of comprehension (*i.e.*, how to "visualize" the infinite state space spanned by an arbitrary arrangement), but also for purposes of tracing back causes of unsafe emergent behaviors to specific culprits (*e.g.*, for debugging purposes). [7]

3 The CHAIN Approach

In this section we present the details of our algebraic approach, CHAIN (a system for the *C*anonical *H*omomorphic *A*bstraction of *I*nfinite *N*etwork protocol compositions). Intuitively, CHAIN represents protocol compositions using *strings*; the infinite set of such strings is reduced to a characteristic finite set via reduction relations (*rewrite rules*) which preserve a correctness property. This section discusses the formal structure and properties of these components which give rise to the desirable properties of a CHAIN system (correct abstraction, sufficient expressive power, homomorphism, termination, canonicity of result).

3.1 Arrangements in CHAIN

Let $G = (\mathcal{N}, \mathcal{E})$ be a graph where the members of $\mathcal{N}$ denote agents which will make up our models and $\mathcal{E}$ are directed edges which indicate valid sequences of those agents (that is, if there is an edge from node n_1 to n_2, then n_2 may immediately follow n_1 in a composition). The set of *chains* in G is then simply the set of finite paths in G, which we denote $\mathsf{paths}(G)$.

For the HTTP protocol, G will look like Figure 1. Notice that our definition of *chains* includes sequences which will make up "partial" network setups, *e.g.*, client connected with a series of proxies (but no server) or even an empty

[5]While this paper will focus upon linear compositions of agents, CHAIN also generalizes to more interesting structures such as trees and digraphs.

[6]Many techniques employ a notion of "equivalence classes" [17, 20], these are generally classes within an abstract internal state representation which do not necessarily translate to intuitively useful insights about the structure of the application or system itself.

[7]The above statement should not be taken to imply that model checking is not a useful verification tool. On the contrary, we actually use model checking to verify the canonical representations we derive through our algebraic approach. Rather, our position is that the state space explosion that model checkers must subdue should not be compounded by the explosion resulting from the representation of arbitrary compositions in the state space.

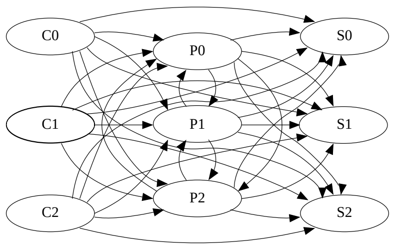

Figure 1: G for HTTP arrangements ($\mathcal{A}$)

sequence. For this reason, we also define the set of *arrangements* as $\mathcal{A} \subseteq \mathsf{paths}(G)$ such that $\mathcal{A}$ are the maximal-length members of $\mathsf{paths}(G)$; for example, all members of $\mathcal{A}$ for the HTTP application are sequences beginning with a single client followed by a sequence of zero or more proxies and terminating with a single server.

One can also imagine more involved systems, such as those in which the composition structure more resembles a tree or a digraph than the sequences represented by $\mathsf{paths}(G)$. CHAIN actually deals principally in rewriting sets of strings; since such sets can easily be used to encode multi-path structures, CHAIN can also be used to effectively "rewrite" such graphs by rewriting their constituent paths.

3.2 Arrangement Properties

We are interested in identifying the members of $\mathcal{A}$ which satisfy (or fail to satisfy) desirable properties, *e.g.*, those that are *deadlock-free*. Let π denote such a property, which can be viewed as a boolean-valued function $\pi : \mathcal{A} \rightarrow \{\textbf{true}, \textbf{false}\}$. Our primary methodological goal is to obtain a "friendly" specification of the two sets:

$$\mathcal{A}_{\textbf{true}} = \{a \in \mathcal{A} \mid \pi(a) = \textbf{true}\}$$

and

$$\mathcal{A}_{\textbf{false}} = \{a \in \mathcal{A} \mid \pi(a) = \textbf{false}\}.$$

By "friendly" we mean, at a minimum, there is a feasible computation to determine whether $a \in \mathcal{A}_{\textbf{true}}$ or $a \in \mathcal{A}_{\textbf{false}}$ for any member of $\mathcal{A}$; ideally, we would also like to devise an easy-to-understand formalism to describe $\mathcal{A}_{\textbf{true}}$ and $\mathcal{A}_{\textbf{false}}$, which can be used to quickly (in polynomial time or better) test whether $a \in \mathcal{A}_{\textbf{true}}$ or $a \in \mathcal{A}_{\textbf{false}}$. We will devise such tests below in Sections 4.1.2 and 4.3.

3.3 CHAIN Reductions

Consider a graph G as described above and a property π on the set $\mathcal{A}$ of arrangements in G. We denote the powerset of a set S by 2^S. We extend $\pi : \mathcal{A} \rightarrow \{\textbf{true}, \textbf{false}\}$ to a function $\pi : 2^{\mathcal{A}} \rightarrow \{\textbf{true}, \textbf{false}\}$ by defining[8] for every $A \in 2^{\mathcal{A}}$:

$$\pi(A) = \begin{cases} \textbf{true} & \text{if } \pi(a) = \textbf{true} \text{ for every } a \in A, \\ \textbf{false} & \text{if } \pi(a) = \textbf{false} \text{ for some } a \in A. \end{cases}$$

Let $\mathcal{A}'$ be some subset, not necessarily proper, of the set $\mathcal{A}$ of arrangements in G. Because $\mathcal{A}$ is a subset of $\mathsf{paths}(G)$, so is $\mathcal{A}'$ a subset of $\mathsf{paths}(G)$. A *reduction function on* $\mathcal{A}'$ is a function $f : \mathsf{paths}(G) \rightarrow 2^{\mathsf{paths}(G)}$ satisfying two conditions:

Invariance on $\mathcal{A}'$: For every $a \in \mathcal{A}'$, it is the case that $\pi(f(a))$ is defined and $\pi(f(a)) = \pi(a)$.

Progress on $\mathcal{A}'$: $$\left(\bigcup_{a \in \mathcal{A}'} f(a)\right) \subsetneq \mathcal{A}'.$$

We can extend $f : \mathsf{paths}(G) \rightarrow 2^{\mathsf{paths}(G)}$ to a function $f : 2^{\mathsf{paths}(G)} \rightarrow 2^{\mathsf{paths}(G)}$ by setting $f(A) = \bigcup_{a \in A} f(a)$ for every $A \in 2^{\mathsf{paths}(G)}$. Thus, the *progress* condition above can be expressed more succinctly as $f(\mathcal{A}') \subsetneq \mathcal{A}'$.

Informally, the *invariance* condition says that π is an invariant of the transformation from $a \in \mathcal{A}'$ to $f(a) \subset \mathcal{A}'$. In practice, this means that, in order to test whether $a \in \mathcal{A}'$ satisfies property π, it suffices to test whether every $b \in f(a)$ satisfies π; as a rule, a desirable reduction is one in which the aggregate of the latter tests is "easier" computationally than the former test.

The *progress* condition is assurance that we gain something by carrying out the transformation from $a \in \mathcal{A}'$ to $f(a) \subset \mathcal{A}'$, *i.e.*, the set $f(\mathcal{A}')$ is a non-empty proper subset of $\mathcal{A}'$. In practice, should $\mathcal{A}'$ be an infinite set we will also want $\mathcal{A}' - f(\mathcal{A}')$ to be an infinite set, *i.e.*, infinitely many arrangements are excluded from the search space $\mathcal{A}'$.

The key insight behind a reduction is that it establishes *behavioral equivalence with respect to* π within some set of chains; a reduction is a statement that "the behaviors of members of set $\mathcal{A}'$ are fully represented by the behaviors of members of its subset $f(\mathcal{A}')$". The means by which this behavioral equivalence is established may be any mechanism appropriate to the given application and π (*e.g.*, logical proofs, type systems [7], process algebra [2], theory of I/O automata [22], I/O equivalence, *etc.*).

3.4 Reduction Strategy

Intuitively, our strategy is to identify a set of reductions (*i.e.*, congruence relations over $\mathcal{A}$ which preserve behavioral equivalence) by which we can establish a finite-sized homomorphic image of $\mathcal{A}$ (that is, a finite-sized $\mathcal{A}_n \subset \mathcal{A}$ such that every member of $\mathcal{A}$ is behaviorally equivalent with members of $\mathcal{A}_n$).

Starting from $\mathcal{A}_0 = \mathcal{A}$, our proposed strategy is to define a nested sequence of strictly decreasing subspaces:

$$\mathcal{A}_0 \supset \mathcal{A}_1 \supset \cdots \supset \mathcal{A}_n$$

[8]Note that other formulations of $\pi(A)$ can be used to bes reflect the semantics of π; *e.g.*, $\pi(A)$ could be the logical *OR* rather than the logical *AND* of all $\pi(a)$ if that better captures the meaning of π. A partial function could also be used, in which $\perp$ is a result value designating an uncertain or undefined result.

induced by a sequence of appropriately defined functions $g_1, g_2, \ldots, g_n$ where $g_i : \mathsf{paths}(G) \to 2^{\mathsf{paths}(G)}$ is derived from a reduction function on $\mathcal{A}_{i-1}$ (or the corresponding set-reduction function) and $\mathcal{A}_i = g_i(\mathcal{A}_{i-1})$ for every $1 \leqslant i \leqslant n$. If successful, this strategy produces a finite search space $\mathcal{A}_n$ such that

$$\mathcal{A}_n = g_n(\cdots(g_2(g_1(\mathcal{A})))\cdots)$$

which implies that for every $a \in \mathcal{A}$

$$\mathcal{A}_n \supseteq g_n(\cdots(g_2(g_1(a)))\cdots)$$

and

$$\pi(a) = \pi(g_n(\cdots(g_2(g_1(a)))\cdots)).$$

3.5 Practical Specification of Reductions

A second methodological goal of our study is a formulation of reduction functions which are both easy to understand and easy to apply in practice. We propose a single framework to simultaneously achieve these two goals—this one and the one mentioned in Section 3.2—by borrowing ideas from algebraic notions of term-rewriting and by using standard techniques for manipulating sets and regular expressions.

Consider some $G = (\mathcal{N}, \mathcal{E})$ as defined above. Let Var be a countable infinite set of formal variables; we use the letters x, y and z (possibly decorated) to denote members of Var. Let $\Sigma = \mathcal{N} \cup \mathsf{Var}$. We use the letters X, Y and Z (possibly decorated) as metavariables ranging over the set Σ^*. If $X \in \Sigma^*$, the set of formal variables occurring in X is denoted $\mathsf{Var}(X)$.

We introduce a particular notion of *rewrite rules*. Each such rewrite rule R will be specified by an expression of the form

$$R : \; X \; \rhd \; \{Y_1, \ldots, Y_n\}$$

satisfying the following conditions:

- $n \geqslant 1$, *i.e.*, the right-hand side is a non-empty finite set,
- $X, Y_1, \ldots, Y_n \in \Sigma^+$, and
- $\mathsf{Var}(Y_1) \cup \cdots \cup \mathsf{Var}(Y_n) \subseteq \mathsf{Var}(X)$.

An *interpretation of* Var (for the given G) is simply a function $\rho : \mathsf{Var} \to \mathsf{paths}(G)$, which is lifted to a function $\bar{\rho} : \Sigma^* \to \mathsf{paths}(G)$ by induction in the obvious way:

1. $\bar{\rho}(\varepsilon) = \varepsilon$,
2. $\bar{\rho}(X\,N) = \bar{\rho}(X)\,N$,
3. $\bar{\rho}(X\,x) = \bar{\rho}(X)\,\rho(x)$,

where $X \in \Sigma^*$, $N \in \mathcal{N}$, and $x \in \mathsf{Var}$. We use ε to denote the empty string.

Let $a, b_1, \ldots, b_n \in \mathsf{paths}(G)$. We say *$a$ rewrites to the set $\{b_1, \ldots, b_n\}$*, using rule R in one step, which we express as:

$$a \rhd_R \{b_1, \ldots, b_n\},$$

just in case there is an interpretation $\rho : \mathsf{Var} \to \mathsf{paths}(G)$ such that $\bar{\rho}(X) = a$ and $\bar{\rho}(Y_i) = b_i$ for every $1 \leqslant i \leqslant n$.

A rewrite rule R as described above induces a function $f_R : \mathsf{paths}(G) \to 2^{\mathsf{paths}(G)}$ as follows. For every $a \in \mathsf{paths}(G)$, we define:

$$f_R(a) = \begin{cases} \{a\} & \text{if } a \not\rhd_R B \text{ for all finite } B \subset \mathsf{paths}(G), \\ \bigcup\{B \subset \mathsf{paths}(G) \mid a \rhd_R B\} & \text{otherwise.} \end{cases}$$

Following standard notation, we write $f_R^{(0)}(a) = \{a\}$ and $f_R^{(k+1)}(a) = f_R(f_R^{(k)}(a))$ for all $k \geqslant 0$. We also define the function $f_R^{(*)} : \mathsf{paths}(G) \to 2^{\mathsf{paths}(G)}$ as follows. For every $a \in \mathsf{paths}(G)$:

$$f_R^{(*)}(a) = \begin{cases} f_R^{(k)}(a) & \text{if there exists } k \geqslant 0 \text{ such that} \\ & \quad f_R^{(k+1)}(a) = f_R^{(k)}(a), \\ & \quad \text{where } k \text{ is the least such,} \\ \text{undefined} & \text{if no such } k \geqslant 0 \text{ exists.} \end{cases}$$

Informally, $f_R^{(*)}(a)$ returns a fix-point of f_R obtained by repeated application of f_R to a, if it exists.

Now, consider the set $\mathcal{A}$ of arrangements in G, a property π on $\mathcal{A}$, and some subset $\mathcal{A}' \subseteq \mathcal{A}$. We say that the rewrite rule R is a *reduction on* $\mathcal{A}'$ provided that the function $f_R^{(*)} : \mathsf{paths}(G) \to 2^{\mathsf{paths}(G)}$ induced by R is a *reduction on* $\mathcal{A}'$ satisfying the two conditions defined in Section 3.3: *invariance* on $\mathcal{A}'$ and *progress* on $\mathcal{A}'$.

Our rewrite rules will satisfy a pleasant condition guaranteeing that $f_R^{(*)}(a)$ is always defined. Let us say that the rule R is *bounded-monotonic in* M iff for some metric M with a minimum value and for all $a, b_1, \ldots, b_n \in \mathsf{paths}(G)$ such that $a \rhd_R \{b_1, \ldots, b_n\}$,

$$\mathsf{M}(a) > \mathsf{M}(b_1)\;,\;\ldots\;,\;\mathsf{M}(a) > \mathsf{M}(b_n).$$

Lemma 3.1. *If the rewrite rule R is bounded-monotonic then, for every $a \in \mathsf{paths}(G)$, it holds that $f_R^{(*)}(a)$ is defined, and is a non-empty finite subset of* $\mathsf{paths}(G)$.

The simplest choice for such a metric is string length (which has a minimum value of zero); where a rule is not *length-decreasing*, it must be shown to be bounded-monotonic in some other metric.

3.6 Confluence of CHAIN Reductions

As with any term re-writing system, at least two properties are important to establish: *termination* (*i.e.*, every arrangement can be rewritten only finitely many times) and *confluence* (*i.e.*, if a can be rewritten to b_1 and b_2, then further rewriting of both of those can produce a single string c). Termination ensures that the system draws some conclusion, while confluence demonstrates the system's internal consistency; the combination of these two properties implies that the final result of the rewriting process is canonical, and that the rewriting process encodes an *equivalence* *class* for each possible result. Establishing confluence also helps to minimize the size of the result set.

A set of rewrite rules terminate if the effects of all rules are *bounded-monotonic* with respect to a single metric; for example, if all rules are *length-decreasing* the a system of such rules must terminate. By Newman's Lemma [1], we know that a rewrite system which terminates is confluent iff it is locally confluent (*i.e.*, given arrangement a which can be rewritten in one step to B_1 or to B_2, both B_1 and B_2 can be rewritten in zero or more steps to B_3). We therefore need only determine local confluence between pairs of rewrite rules in order to establish confluence and (thereby) the canonicity of the terminal result of rewriting any arrangement.

In the simplest cases, a pair of rewrite rules R_i and R_j are locally confluent if they are commutative, *i.e.*, $f_i^{(*)}(f_j^{(*)}(a)) \equiv f_j^{(*)}(f_i^{(*)}(a))$ for all $a \in \mathcal{A}$. As much as possible, we rely upon this property in our local confluence proofs.[9]

It should be noted that a non-confluent set of rewrite rules is not a failure of our methodology. In those systems where rewrite rules represent *behavioral equivalence*, rewrites embody a transitive property; it follows that any such divergence identifies additional equivalence relationships which can give rise to valid rewrite rules.

As such, any arrangement $a \in \mathcal{A}$ for which two rewrite rules R_i and R_j provide diverging evaluation paths actually becomes an instance of a proper behavioral equivalence relation; $f_i(a)$ and $f_j(a)$ are behaviorally equivalent sets (because of the *Invariance* property), and as such if either set has only one member, the pair can be directly transformed into a previously unknown reduction. Such additional relations resolve the failures of local confluence, but must then be tested for interference with each other and the original rule set; this can be performed mechanically using the Knuth-Bendix procedure [18].

3.7 Sufficient Subspaces

If $f_R^{(*)}$ is always defined, then the application of $f_R^{(*)}$ to all members of a language $\mathcal{A}$ will yield some subset of $\mathcal{A}$ such that, for every $a \in \mathcal{A}$, the value of $\pi(a)$ can be easily determined from $\pi(f_R^{(*)}(a))$. For this reason, we refer to the output $f_R^{(*)}(\mathcal{A})$ as a *sufficient subspace*.

In the rest of the paper, when there is no ambiguity, notions that have been defined for a rewrite rule R are extended to the function $f_R^{(*)}$ in the obvious way; for example, we say "$f_R^{(*)}$ is length-decreasing" if R is length-decreasing.

It is also convenient to introduce the notion of the *support* of the function $f_R^{(*)}$, or of its associated rewrite rule R:

$$\mathsf{support}(f_R^{(*)}) = \mathsf{support}(R) = \{a \in \mathcal{A} \mid f_R^{(*)}(a) \neq \{a\}\},$$

i.e., $\mathsf{support}(f_R^{(*)})$ is the portion of $\mathcal{A}$ on which $f_R^{(*)}$ acts non-trivially; so,

$$f_R^{(*)}(\mathcal{A}) = (\mathcal{A} - \mathsf{support}(f_R^{(*)})) \cup f_R^{(*)}(\mathsf{support}(f_R^{(*)}))$$

Recall our stated strategy from Section 3.4: to define a nested sequence of strictly decreasing subspaces $\mathcal{A} = \mathcal{A}_0 \supset \mathcal{A}_1 \supset \cdots \supset \mathcal{A}_n$ induced by a sequence of reduction functions $f_1, \ldots, f_n$. In what follows, for every $1 \leqslant i \leqslant n$, we use f_i to denote the $f_{R_i}^{(*)}$ induced by the bounded-monotonic rewrite rule R_i.

A trivial approach to this goal is to find $f_n(\cdots(f_2(f_1(\mathcal{A})))\cdots)$. However, such a formulation fails to properly account for *convolutions* of rewrite rules. For example, consider a system in which R_1 reduces all sequences of "a"s to a single "a" and R_2 removes any "b" appearing immediately between two "a"s; for the string aba, clearly $f_2^{(*)}(f_1^{(*)}(\texttt{aba})) \neq f_1^{(*)}(f_2^{(*)}(f_1^{(*)}(\texttt{aba})))$; the convolution of f_1 and f_2 has a greater effect than their sequential application.

Rather than composing the functions as such, we individually consider the effect of each rule upon $\mathcal{A}$, and then take the intersection of the resulting sufficient subsets. The intersection operator allows our examination of each f_i to wholly exclude from future consideration all arrangements reduced away by f_i.

Lemma 3.2. *Consider a set of reductions $\mathcal{R} = \{R_1, \ldots, R_n\}$ inducing functions $\mathcal{F} = \{f_{R_1}^{(*)}, \ldots, f_{R_n}^{(*)}\}$ which are all monotonic with respect to some metric. A sufficient subspace $\mathcal{A}_n$ can be defined inductively where $0 < n$ and $\mathcal{A}_0 = \mathcal{A}$ as:*

$$\mathcal{A}_n = \mathcal{A}_{n-1} \bigcap f_{R_i}^{(*)}(\mathcal{A})$$

or directly as:

$$\mathcal{A}_n = \bigcap_{i=1}^{n} \left(f_{R_i}^{(*)}(\mathcal{A}) \right) \tag{1}$$

which can be equivalently stated as:

$$\mathcal{A}_n = \bigcap_{i=1}^{n} \left(\mathcal{A} - \mathsf{support}(f_{R_i}^{(*)}) \right) \cup f_{R_i}^{(*)}(\mathsf{support}(f_{R_i}^{(*)})) . \tag{2}$$

Where the set of reductions is clear from context and where $\mathcal{A}_n$ (the sufficient subspace which is most reduced with respect to the given reductions) is of finite ordinality, we refer to it as $\mathcal{A}_\top$.

Corollary 3.2.1. $\pi(\mathcal{A}) = \pi(\mathcal{A}_\top)$.

Intuitively, this corollary tells us that if we can prove some property (*e.g.*, freedom from deadlocks) for all members

[9] This is a special case of the general definition of local confluence: A pair is locally confluent if there exist compositions (call them F_1 and F_2) of f_is drawn from the full set of valid rewrite rules ($\mathcal{R}$) such that $F_1(f_i^{(*)}(a)) \equiv F_2(f_j^{(*)}(a))$ for all $a \in \mathcal{A}$. Intuitively, this means two rules are confluent if the divergence they introduce can be reconciled by any sequences of additional rewrites.

Standard	Client	Proxy	Server
RFC1945 (HTTP/1.0)	C0	P0	S0
RFC2068 (obsolete HTTP/1.1)	C1	P1	S1
RFC2616 (HTTP/1.1)	C2	P2	S2

Table 1: HTTP Agent Models

Rewrite Rule	A_i, *i.e.*, $f_i(\mathcal{A})$
x P0 y $\rhd_{R_1}$ { x S0, C0 y }	$\mathcal{A} - \mathcal{C}\mathcal{P}^*$ P0 $\mathcal{P}^*\mathcal{S}$
x P0 P0 y $\rhd_{R_2}$ { x P0 y, C0 S0 }	$\mathcal{A} - \mathcal{C}\mathcal{P}^*$ P0 P0 $\mathcal{P}^*\mathcal{S}$
C2 P2 x $\rhd_{R_3}$ { C2 x }	$\mathcal{A} -$ C2 P2 $\mathcal{P}^*\mathcal{S}$
x P2 S2$\rhd_{R_4}$ { x S2 }	$\mathcal{A} - \mathcal{C}\mathcal{P}^*$ P2 S2
x P2 P2 y $\rhd_{R_5}$ { x P2 y }	$\mathcal{A} - \mathcal{C}\mathcal{P}^*$ P2 P2 $\mathcal{P}^*\mathcal{S}$
x P1 P1 P1 y $\rhd_{R_6}$ { x P1 P1 y }	$\mathcal{A} - \mathcal{C}\mathcal{P}^*$ P1 P1 P1 $\mathcal{P}^*\mathcal{S}$
C0 P1 x $\rhd_{R_7}$ { C1 x }	$\mathcal{A} -$ C0 P1 $\mathcal{P}^*\mathcal{S}$
x P1 P2 P1 y $\rhd_{R_8}$ { x P1 P1 y }	$\mathcal{A} - \mathcal{C}\mathcal{P}^*$ P1 P2 P1 $\mathcal{P}^*\mathcal{S}$
C1 P1 P1 x $\rhd_{R_9}$ { C1 P1 x }	$\mathcal{A} -$ C1 P1 P1 $\mathcal{P}^*$ $\mathcal{S}$
C1 P2 P1 x $\rhd_{R_{10}}$ { C1 P1 x }	$\mathcal{A} -$ C1 P2 P1 $\mathcal{P}^*$ $\mathcal{S}$

Table 2: Rewrite rules $(R_1, \ldots, R_{10})$ and their resulting sufficient subspaces $(f_1(\mathcal{A}), \ldots, f_{10}(\mathcal{A}))$

of the minimal sufficient subset (*i.e.*, $\{\pi(a) = \textbf{true} \,|\, a \in \mathcal{A}_\top\}$), then we have also proven that same property for all members of the original (infinite) set $\mathcal{A}$.

4 Example Applications of CHAIN

In this section, we exemplify the application of the CHAIN approach to a series of interesting network protocol correctness problems. We begin with a detailed completion of our example of HTTP request continuation deadlock-safety in order to clearly illustrate the workings of CHAIN. We then proceed to a more abbreviated discussion of its application to the analysis of an applet for selective dropping of MPEG frames in an overlay network. Finally, we sketch its application to a web intra-cache consistency algorithm.

4.1 HTTP Deadlock-Safety

Through careful study of our models of HTTP protocol agents, we have derived and proven a set of rewrite rules which preserve the behavior of chains with respect to HTTP request continuation. If an arrangement is deadlock-prone, then any arrangement which can be rewritten to that one will also be deadlock-prone; likewise, any arrangement to which it can be rewritten will also be deadlock-prone. The same holds for arrangements which are deadlock-free. The derivation of these and other rules is discussed in greater depth in [6]. Eight of the rules derived there pertaining to our current goal are presented in Table 2, along with two more rules (R_9 and R_{10}), the derivation of which will be discussed below.

In this paper, we refer to particular models using the letter-number pairs presented in Table 1; these represent the favored models for each revision/role.

All of the listed rules are proper rewrite rules as defined in Section 3.5, satisfying the three necessary conditions: (1) each has a non-empty right-hand side; (2) all strings are members of Σ^+; and (3) all $\mathsf{Var}(Y_i) \subseteq \mathsf{Var}(X)$. Notice that these rules are also all length-decreasing, which implies (by Lemma 3.1) that $f^{(*)}_{R_i}$ is always defined for all of them. Therefore, each of the preceding rewrite rules R_i gives rise to a function $f^{(*)}_{R_i}$, henceforth denoted by f_i. The support of f_i, and the set to which f_i maps its support, are presented in Table 2 using regular expressions. For brevity, $\mathcal{C} = (\mathrm{C0} \mid \mathrm{C1} \mid \mathrm{C2})$, $\mathcal{P} = (\mathrm{P0} \mid \mathrm{P1} \mid \mathrm{P2})$, and $\mathcal{S} = (\mathrm{S0} \mid \mathrm{S1} \mid \mathrm{S2})$.

Notice also that each f_i (*i.e.*, each $f^{(*)}_{R_i}$) is a valid reduction function, in that it satisfies the *invariance* and the *progress* properties. Invariance was previously established; progress holds because for every f_i it is true that $f_i(\mathcal{A}) \subsetneq \mathcal{A}$.

4.1.1 Confluence

We next ask whether this set of reductions is confluent, *i.e.*, whether every arrangement $a \in \mathcal{A}$ will be terminally rewritten to a single result set independent of the reduction strategy. Because our set of rewrite rules will terminate (because they are all length-decreasing), this is equivalent to asking if the set of reductions is locally confluent.

The local confluence of most pairs of reductions is straightforward to see, because the rules are independent. Clearly R_1 and R_2 operate upon chains which no other reductions operate upon; the cluster of R_3, R_4, and R_5 likewise are clearly independent in their effects of the predicate chains of R_1, R_2, R_6, R_7, and R_8, and similarly the cluster of R_6, R_7, and R_8 are independent of the first five rules. So our only concern is local confluence within these three clusters.

R_1 **and** R_2**:** Since R_2 is an instantiation of R_1, these two clearly do not lead to contradictory rewrite strategies.

R_3**,** R_4 **and** R_5**:** All three of these rules remove P2 from chains. Consider the one case where both R_3 and R_4 apply to the same P2, namely, $a =$ C2 P2 S2. f_3 and f_4 remain commutative, because $f^{(0)}_{R_3}(f^{(1)}_{R_4}(a)) = f^{(0)}_{R_4}(f^{(1)}_{R_3}(a))$. Similarly, consider the set of chains over which R_3 and R_5 conflict, namely, $a =$ C2 P2 P2 x for any x; f_3 and f_5 are clearly commutative because $f^{(0)}_{R_3}(f^{(1)}_{R_5}(a)) = f^{(0)}_{R_5}(f^{(1)}_{R_3}(a))$ when f_5 affects the specified subchain of a. The same proof holds for the pairing of R_4 and R_5. Since all three pairings of these three rewrites commute, the set is locally confluent.

R_6**,** R_7 **and** R_8**:** While rules R_6 and R_8 are clearly independent in their effects, the other two pairs within this cluster are not commutative.

- R_6 and R_7: These rewrite rules diverge on chains of the form C0 P1 P1 P1 x. R_7 rewrites this expression to C1 P1 P1 x, and R_6 rewrites it to C0 P1 P1 x; while the latter can then be rewritten using R_7 to C1 P1 x, they still identify different sets, and there exists no rewrite strategy which will (for all values of x) rewrite both of these to a common expression.

- R_7 and R_8: These rewrite rules diverge on chains of the form C0 P1 P2 P1 x. R_7 rewrites this expression to C1 P2 P1 x, while R_8 rewrites it to C0 P1 P1 x. Much as above, the second expression can again be rewritten to C1 P1 x, but from there no rewrite strategy exists to rewrite these to a common expression.

Applying the procedure discussed in Section 3.6, each of these conflicts is transformed into a new valid rewrite rule in a straightforward way so as to preserve the length-decreasing property; the results are rules R_9 and R_{10}, which succeed in rendering the system confluent, so further iteration of the procedure is not necessary.

Theorem 4.1. *The set of rewrite rules $\mathcal{R} = \{R_1, \ldots, R_{10}\}$ is confluent.*

Proof. The local confluence of most rule pairs is already discussed above. The two new rewrite rules are independent of each other and independent of the original eight rules, with the exception that R_9 and R_6 have identical effect upon expressions C1 P1 P1 P1 x and are thus commutative, and that R_{10} and R_6 have identical effect upon expressions C1 P2 P1 P1 P1 x and are thus commutative. Both of the failures of local confluence among $\{R_1, \ldots, R_8\}$ are addressed by application of the two new rules. Therefore, $\mathcal{R}$ is locally confluent.

Because the rules are all length-decreasing (and therefore bounded-monotonic), the system terminates; by Newman's Lemma, it is therefore confluent. ☐

Corollary 4.1.1. *The set of rewrite rules $\mathcal{R} = \{R_1, \ldots, R_{10}\}$ defines a canonical subset of $\mathcal{A}_\top$ for every $a \in \mathcal{A}$,* i.e., *every $a \in \mathcal{A}$ is a member of an equivalence class defined by a subset of $\mathcal{A}_\top$.*

4.1.2 Reducing the Model Space

Recall Equations 1 and 2, which are the "glue" of our strategy. For each f_i we find $f_i(\mathcal{A})$, *i.e.*:

$$(\mathcal{A} - \mathsf{support}(f_i)) \cup f_i(\mathsf{support}(f_i))$$

We denote such a set induced by any f_i as A_i. Using the above-described supports and the sets they are mapped to, Table 2 presents these in simplified regular expression form for each of f_1 through f_{10}.

Taking the intersection of these sufficient subspaces gives us the finite minimal sufficient subspace supported by the given reduction functions. We will use both regular expressions and finite state automata (in the style of the graph G described earlier) to describe such subspaces (sets of strings) for the remainder of this paper as appropriate.

The automaton for $\mathcal{A}$ is a simple nine-state machine with three start states each with no inbound edges (C0, C1, and C2), three end states with no outbound edges (S0, S1, and S2), and three intermediary states (P0, P1, and P2) which are fully connected with one another, each connected to by every start state, and each connecting to every end state. Each start state also connects directly to every end state. This was already illustrated in Figure 1.

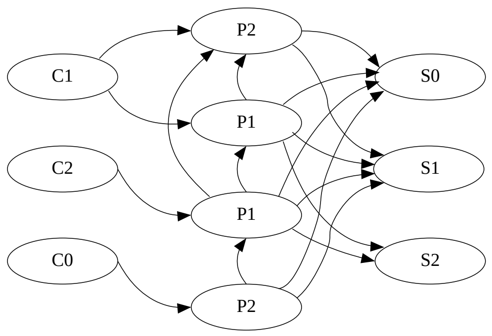

Figure 2: Automaton for $\mathcal{A}_{10} - \mathcal{CS}$, *i.e.*, $A_1 \cap \cdots \cap A_{10} - \mathcal{CS}$

$$((\mathcal{C}\ \mathcal{P}^*\ \text{P1}) \mid \text{C1})\ (\text{P1} \mid \text{P2}^+)\ (\text{S0} \mid \text{P0}\ \mathcal{P}^*\ \mathcal{S})$$
$$((\mathcal{C}\ \mathcal{P}^*\ \text{P1}) \mid \text{C1} \mid \text{C2})\ (\text{P1} \mid \text{P2})^*\ \text{P1}\ (\text{S0} \mid \text{P0}\ \mathcal{P}^*\ \mathcal{S})$$

Figure 3: Deadlock-Prone Arrangements (Theorem 4.3)

The intersection of the ten sets $A_1 \ldots A_{10}$ is presented in Figure 2 as an automaton. (For visual clarity, the $\mathcal{CS}$ edges have been omitted; each client also has an edge to each server.) This represents the minimal sufficient subset of $\mathcal{A}$ under the ten reductions $R_1 \ldots R_{10}$. Of particular significance is that this set is finite (equivalently, the automaton is acyclic, or the regular expression has no unbounded repetitions). As such, we have satisfied the goal of our strategy by identifying a finite $\mathcal{A}_n = A_1 \cap \cdots \cap A_{10}$ which is a sufficient subset of $\mathcal{A}$. This set $\mathcal{A}_{10}$ has 29 member strings; thus, it is sufficient to compute $\pi(a)$ for only these 29 members of $\mathcal{A}$ in order to acquire a trivial procedure for the determination of $\pi(a)$ for any $a \in \mathcal{A}$. We have thus proven the following theorem:

Theorem 4.2. *Let $\mathcal{A}$ be the infinite space of all arrangements of HTTP agents as defined in Section 3.1, and let $\mathcal{R}$ be the set of rewrite rules discussed above which preserve behavioral equivalence. We can construct a finite subset $\mathcal{A}_\top$ of $\mathcal{A}$, consisting of 29 member arrangements, which satisfies the following condition: By the application of $\mathcal{R}$, every $a \in \mathcal{A}$ can be rewritten to a subset B of $\mathcal{A}_\top$ such that a satisfies π if and only if every $b \in B$ satisfies π.*

Proof. Follows directly from Lemma 3.2 and the above analysis. ☐

A π Decision Rule Using Failure Patterns: As alluded to above, we would ultimately like to express a decision rule in a compact and computationally efficient form which will allow us to decide $\{\pi(a) \mid a \in \mathcal{A}\}$. In [5], we happened upon a pair of "failure patterns" which accounted for all failure cases explored in that paper. Here we argue for the correctness of a pair of failure patterns derived from $\mathcal{A}_\top$ and the above reductions–a process that *methodically discovers* all deadlock cases for all arrangements in $\mathcal{A}$.

Theorem 4.3. *All* $a \in \mathcal{A}$ *such that* $\pi(a) =$ **false** *will match a regular expression stated in Figure 3.*

Proof. Among $a \in \mathcal{A}_{10}$, all a such that $\pi(a) =$ **false** (that is, all $a \in \mathcal{A}_{\textbf{false}}$) match at least one of the stated patterns, and no a such that $\pi(a) =$ **true** (*i.e.*, no $a \in \mathcal{A}_{\textbf{true}}$) matches either.

As we have shown, all members of $\mathcal{A}$ which are deadlock-prone are reducible to members of $\mathcal{A}_{10}$ which are deadlock-prone, and similarly, all members of $\mathcal{A}$ which are deadlock-safe are reducible to members of $\mathcal{A}_{10}$ which are deadlock-safe.

As such, the correctness of this theorem rests upon three properties: (1) for any member $a \in \mathcal{A}_{10}$, $\pi(a) =$ **false** iff a is in the union of these patterns (that is, the patterns correctly identify $\mathcal{A}_{\textbf{false}} \cap \mathcal{A}_{10}$); (2) any member of the union of these two patterns is reducible to a member of $\mathcal{A}_{\textbf{false}} \cap \mathcal{A}_{10}$; (3) no arrangement $a \in \mathcal{A}$ which does not match either of these patterns can be reduced to one which does.[10] Item-by-item proofs of these properties are included in Appendix A.

□

4.2 An MPEG Overlay Routing Protocol

While model checking is often applied in *post mortem* fashion to assess bugs and problems in existing protocols and software, this need not be the case. This section presents an application of CHAIN approaching the problem from a design perspective rather than a retroactive analysis perspective.

Many algorithms proposed for overlay networks are, by their nature, designed to be deployed into a network in which they will interact with other applications, as well as conventional routers and hosts (*i.e.*, we expect that they will be composed with other processes, perhaps both controlled and emergent). Thus it seems reasonable to believe that such applications are good candidates for analysis using CHAIN.

Consider the method for handling MPEG flows proposed in [15], which drops MPEG frames based upon its own drop history and the dependency and priority relationships between the three classes of MPEG frames (I, P, and B frames).[11] These relationships suggest simple packet-dropping rules: (1) If at all possible, dropping I frames should be avoided; (2) once a B frame has been dropped, all successor B frames until the next P frame are useless and should be dropped; and similarly (3) once a P frame has been dropped, all successor packets can be safely dropped until the next I frame. The applet presented in [15] implements a simple version of this algorithm requiring constant time and storage.

Unfortunately, the router[12] so described must itself receive all of the packets constituting an MPEG stream in order to behave correctly. These are unreasonably optimistic assumptions [25]; indeed, it is not hard to devise pathological reorderings among pairs of sequentially adjacent packets which can cause the applet to wrongly treat large numbers of frames as "worthless" (and therefore discardable), and if certain packets are dropped before reaching such a router, it can erroneously forward large numbers of worthless packets.

We can easily cast either or both of these concerns (packet ordering and drop-tolerance) in CHAIN. There is nothing intrinsic to a network which drops or reorders packets that prevents it from being represented as an agent in the same sense in which the MPEG router is an agent. Therefore, it makes sense for us to ask within our framework whether the composition of a packet-reordering network with such a router would cause it to behave erroneously (that is, to drop packets which could still be valuable to an end-host)? Does the direct composition of two such routers induce packet drops that a single such router would not? What about composing two such routers using a packet-reordering network, or composing two network-router pairs? What about compositions with other kinds of routers which perform random drops, or which may do retransmissions on their own (*e.g.*, a wireless base station)? All of these questions can be framed in terms of a binary correctness property π which determines whether packets could be erroneously dropped by any particular composition of those components.[13]

4.2.1 Representing and Reducing the Network

Some reductions naturally arise as properties of the basic existing network infrastructure; For example:

$$R_1 : \ x \ \textit{reord reord} \ y \ \rhd \ \{x \ \textit{reord} \ y\}$$

$$R_2 : \ x \ \textit{drop drop} \ y \ \rhd \ \{x \ \textit{drop} \ y\}$$

$$R_3 : \ x \ \textit{reord drop} \ y \ \rhd \ \{x \ \textit{drop reord} \ y\}$$

[10]The second and third properties taken together represent the *closure* of the set described by these expressions under all reductions in $\mathcal{R}$.

[11]MPEG streams are structured as follows: each I frame signifies the beginning of a new group of pictures (GOP). Within a GOP, each P frame can only be decoded if the initial I frame and all previous P frames have been received. Similarly, a B frame can only be decoded if the previous P frame could be decoded and if all B frames between that P frame and itself have been received.

[12]We henceforth use "router" to mean a node in an (overlay) network that handles the forwarding of the packet to one (or more) other nodes in the network.

[13]The MPEG router is itself allowed to drop packets when its downstream link load is too high. This means that, for the purposes of π, whenever the router wants to forward a packet there are three possible outcomes: An overload drop, a correct forwarding, and a "legal" drop. It is not appropriate for this question to consider a notion of "how many" drops are acceptable - a drop is either inevitable because of link overload (which may arise at arbitrary times), correct (because previous drops render the packet worthless), or illegal (the packet was still valuable and the link was not overloaded). A different π property could be defined and examined to include a notion of drop rate, or bounded delay and jitter, or anything else of interest.

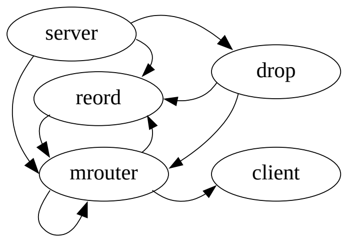

Figure 4: $\mathcal{A}_4$ for basic MPEG-routing network

Notice that the third rule forces a particular ordering upon adjoining *reord* and *drop* nodes; in so doing, these three rules together form a compound rule which says that any chain consisting only of both *drop* and *record* can be represented with a single canonical chain, "*drop reord*".[14]

We also note that the *mrouter* node will always be permitted to drop packets because of internal congestion; thus, another rule will always be applicable[15]:

R_4 : x *mrouter drop* y $\triangleright$ $\{x$ *mrouter* $y\}$

Ultimately, we would like to be able to say something about the *mrouter* in any network arrangement. Assuming that all the relevant characteristics of intervening networks can be represented using models of *record* and *drop*, we can then define $\mathcal{A}$ as $\mathcal{SP}^*$ *mrouter* $\mathcal{C}$, where

$$\begin{aligned}\mathcal{S} &= \textit{server}\\ \mathcal{P} &= \textit{mrouter}, \textit{reord}, \textit{drop}\\ \mathcal{C} &= \textit{client}\end{aligned}$$

This definition handles two simplifications of the problem space for us up front: it excludes all arrangements which do not include at least one *mrouter* (*i.e.*, all arrangements in which we are not interested) and it removes all *reord* or *drop* which do not precede an *mrouter* (because they have no bearing upon the correctness of an *mrouter*). Given the already-stated four reductions, we can derive $\mathcal{A}_4$ (pictured in Figure 4) using the corresponding $f_1, \ldots, f_4$. $\mathcal{A}_4$ clearly still represents an infinite set of arrangements (or language), so our methodology will require additional reductions in order to produce useful results.

Reducibility as Specification: Rather than thinking in terms of the reductions which a particular fully-specified application induces, one may prefer to state a design goal for an application in terms of a set of reductions which that application must satisfy. For example, we could state as a design property that a sequence of identical agents will be behaviorally equivalent to a single such agent; any protocol or implementation which disagrees with this property will fail to meet the design constraints.

We can then state, as an engineering goal, that *mrouter* should provably satisfy a set of reductions which yield a finite $\mathcal{A}_n$. The choice of proof method is not particularly important; whichever is best suited to the design and development environment can equally well be used. The following reductions would be sufficient, and make for an illustrative example of target reductions which could be set as correctness criteria for some *mrouter* formulation:

R_5 : x *mrouter mrouter* y $\triangleright$ $\{x$ *mrouter* $y\}$

R_6 : x *reord mrouter reord mrouter* y $\triangleright$
$\{x$ *drop reord mrouter* $y\}$

If all six reductions are valid, then $\mathcal{A}_6$ has six members; thus, by testing only those six, we establish the behavior of *mrouter* in all possible network configurations. $\mathcal{A}_6$ is expressed by the automaton in Figure 5.

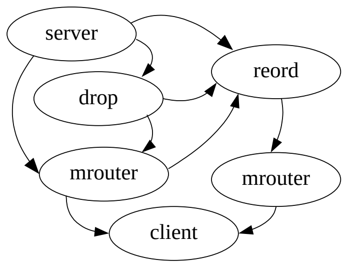

Figure 5: Acyclic automaton of $\mathcal{A}_6$ for an ideal MPEG router

Confluence: Proof that this system of rewrite rules is confluent (and therefore gives rise to a canonical form) is straightforward.

Theorem 4.4. *The set of rewrite rules $\mathcal{R} = \{R_1, \ldots, R_6\}$ is confluent.*

Proof. We begin by proving termination. All rules but R_3 are length-decreasing (and therefore monotonic, so their corresponding $f^{(*)}$'s are defined); R_3 does not increase length and is clearly itself monotonic because it always moves *drop* nodes to the left and *reord* nodes to the right; the rules are clearly monotonic with respect to a composition of these two metrics (length as the major and *drop*/*reord* ordering as the minor component), thus $\mathcal{R}$ terminates. We can therefore establish its confluence using Newman's Lemma by demonstrating local confluence.

All pairings among R_1, R_2, R_4, R_5, and R_6 are clearly commutative over any chains in which their effects overlap. R_3 is similarly commutative with R_4, R_5, and R_6. This leaves only the pairings of R_3 with R_1 and R_2.

Consider the chain *reord reord drop*. By R_1 it is rewritten to *reord drop*; by R_3 it is rewritten to *reord drop reord*.

[14]This is where exclusion of reductions becomes useful; while the inverse of R_3 is also a true proposition, its co-introduction into a system along with R_3 would introduce a rewrite cycle between itself and R_3, which allows the existence of a non-terminating rewrite strategy and definitionally precludes the existence of a "canonical" reduction for each arrangement.

[15]Notice that a similar rule which removes *drop* from before *mrouter* in a chain would also be valid; however, we do not introduce this rule, as it would interfere structurally with other rules, requiring a much more involved set of rules to keep the system confluent.

Notice that these are indeed confluent: the former can then be rewritten by R_3 to *drop reord* and the later can be rewritten by R_3 (again) to *drop reord reord*, which can be rewritten by R_1 to *drop reord*. A similar case arises when R_3 is paired with R_2. Therefore, the lack of commutativity within these pairs is resolved by the existence of a succeeding rewrite strategy which brings their results into agreement.

Since $\mathcal{R}$ is both locally confluent and terminating, it is therefore confluent. □

It is easy to devise variations upon this $\mathcal{R}$ which express the same properties of the problem space but are difficult to make confluent[16]; *e.g.*, an additional rule (based upon the rationale of R_4) which removes a *drop* preceding an *mrouter* is non-commutative with R_3, and converting this conflict into an additional rewrite rule does not result in confluence; many iterations of the Knuth-Bendix procedure are required to resolve this divergence. As another example, if R_3 is replaced with its inverse, a similar conflict arises between it and R_6, and the simplest apparent solution (resolved by including both R_3 and its inverse) does not work because the pair support allow a non-terminating rewrite strategy.

4.2.2 Algorithms Resilient to Network Anomalies

We now discuss modified versions of the algorithm from [15] which are more resilient to common network anomalies. We use a modification of their proposed packet header which adds an integer field, "`dep_frame_no`". In an I-frame, this field has the same value as `frame_no`. In a P-frame, `dep_frame_no` is the `frame_no` of the previous P-frame within the GOP, or the `frame_no` of the previous I-frame if this is the first P-frame in the GOP. For a B-frame, `dep_frame_no` is the `frame_no` of the previous P-frame. This field acts to make explicit and unambiguous the dependency relationships among packets.

For the purposes of this section, we will use the words "frame" (in the MPEG sense) and "packet" (in the IP routing sense) interchangeably.

Missing Packets: If the packet stream remains in order but may lose packets before reaching the *mrouter*, then we can easily implement an aggressive drop algorithm. In addition to the new header field, we require that each "movie" record include the `frame_no` of the last frame forwarded as part of that movie. The arrival of a packet whose `frame_no` is greater than the previous `frame_no` plus one indicates packet loss; the response depends upon the type of the arriving packet and the (inferred) types of (some) dropped packets. Pseudocode for an algorithm is given in Figure 6.

Essentially, we have used the `dep_frame_no` field, taken together with the frame type, to deduce the type of the missing packet (or that of the most important packet within a gap) and to choose a drop strategy accordingly.

```
Require: packet p
  if p is I-frame then
    drop_PB_mode = drop_B_mode=false
    forward p
    last_frame_no = p.frame_no
  else if p is P-frame then
    if drop_PB_mode==true then
      drop p
    else
      if last_frame_no < p.dep_frame_no then
        drop p
        drop_PB_mode = drop_B_mode = true
      else
        forward p
        drop_B_mode=false
        last_frame_no = last_pframe_no = p.frame_no
      end if
    end if
  else if p is B-frame then
    if drop_B_mode==true then
      drop p
    else
      if last_pframe_no < p.dep_frame_no then
        drop p
        drop_PB_mode = drop_B_mode = true
      else if last_frame_no < p.frame_no - 1 then
        drop p
        drop_B_mode = true
      else
        forward p
        last_frame_no = p.frame_no
      end if
    end if
  end if
```

Figure 6: Algorithm resilient only to missing packets

This algorithm would precede all local policy decisions, including congestion-driven drops; acting as such, it prevents "worthless" packets from reaching the overlay routing logic.

Using this implementation of the *mrouter*, we can easily prove target reduction R_5, as the second *mrouter* will in all cases pass through packets forwarded by the first. Proving R_6 is more involved; we omit it for want of space[17]. Since all rewrite rules in $\mathcal{R}$ apply for this algorithm, CHAIN allows us to explore its correctness under all possible networks using only the six arrangements in $\mathcal{A}_\top$ (the homomorphic image of all arrangements).

Reordered Packets: For each movie, we add three ring buffers acting as "policy drop logs": one for I-frames, one for P-frames, and one for B-frames (`drop_I_log`, `drop_P_log`, and `drop_B_log`, respectively). Each

[16]Note that this does not contradict Knuth-Bendix or our claims in Section 3.6; this is a statement about the simplicity and compactness of the set of rewrite rules, not about the ability of the system to be made confluent.

[17]A rough sketch of the proof is as follows: because the algorithm is run before congestion can induce drops within the mrouter, it is unaware of such drops, so any subset of a packet sequence can be dropped by a *reord mrouter* chain; therefore, that chain can be represented to its peers as *reord drop*. If we use this to rewrite the first such subchain in the predicate, we get *drop reord reord mrouter* which (by R_1) is equivalent to *drop reord mrouter*. We must also capture the behaviors of the initial *reord mrouter* in case they would induce an illegal drop; since one of the behaviors of *drop reord mrouter* captures exactly that, all behaviors of the predicate are captured by the consequent, *ergo* they are behaviorally equivalent. □

```
Require: packet p
Require: congestion drops noted in drop_*_log
  if p is I-frame then
    forward p
  else if p is P-frame then
    if dep_frame_no $\in$ drop_P_log $\cup$ drop_I_log then
      drop p
      note p.frame_no in drop_P_log
    else
      forward p
    end if
  else if p is B-frame then
    if p.dep_frame_no $\in$ drop_P_log then
      drop p
      note p.frame_no in drop_B_log
    else if $\exists$ z$\in$ drop_B_log s.t. p.dep_frame_no $<$ z $<$ p.frame_no then
      drop p
      note p.frame_no in drop_B_log
    end if
  end if
```

Figure 7: Algorithm resilient only to reordering

backlog has a fixed size N_I, N_P, N_B; setting all of these to one will (naturally) minimize the running time and space requirements of the algorithm, but using larger values of N makes the algorithm more robust to larger delay/reordering spans.

The premise of this proposed algorithm is that we will only drop packets which we know for certain are worthless, that is, which we can prove to be so using our retained state. Since frames may arrive out-of-order, a simple gap in sequence numbers is not sufficient to infer loss; stronger proof is required. Specifically, we will only make dependency-drops based upon our own drops driven by internal congestion or previous policy choices. This is formalized by the algorithm in Figure 7.

This algorithm guarantees that the *mrouter* does not drop any packet which will still be valuable to the client; it does this at the expense of more per-movie state (linear in N), more per-frame computation (linear in N), and a less aggressive drop policy than the naïve original algorithm (it may keep packets the original would have dropped). A larger N will allow for a more aggressive drop policy; it could actually prove to *correctly* drop more packets than the original algorithm under certain orderings (as contrasted with the incorrect drops and incorrect retentions which the original could induce).

Unlike the drop-resilient algorithm, this algorithm should retain separate drop logs for each output channel within the overlay; it would therefore make the most sense for it to be instantiated after the overlay routing logic on a per-output-queue basis. This requires a simple augmentation to the routing logic itself in order to make note of drops[18] in the log arrays (probably implemented as ring buffers).

For this reordering-resilient *mrouter*, R_6 can be easily proven, since *mrouter* can drop any subset of a stream if an appropriate reordering is fed to it by the preceding *reord*.

[18]Perhaps such drops will be controlled by the filling of a send buffer, the closing of a peer's advertised window, or by some congestion-control policy of the overlay algorithm itself.

R_5, however, only holds in a general sense: recalling that *mrouter* is parameterized with the length of the drop logs, a pair of *mrouter*s each with logs of length n are behaviorally equivalent to a single *mrouter* with logs of length $2n$.[19] Since (if we allow for this condition) all rules in $\mathcal{R}$ apply for this algorithm, again CHAIN allows us to quantify its correctness in all possible networks using only the arrangements in $\mathcal{A}_\top$.

Resilience to Both Reordered And Missing Packets: There is a fundamental difficulty in trying to handle both of the above network anomalies aggressively in a single, fixed-state algorithm: a delayed packet is indistinguishable from a dropped packet until it arrives. It is therefore impossible to differentiate the cause of a sequence gap unless we queue packets and "release" them when the necessary dependence packets arrive (or drop them when some threshold number of packets have passed without the dependency's arrival). Essentially, by the time we could infer with high confidence that a packet has been dropped, too many other packets will have already arrived, the storage of which is too costly for a reasonable minimal-state algorithm (particularly considering that the stored packets may turn out to be worthless). We therefore offer no such applet for analysis here.

4.3 Web Intra-Cache Consistency

There is nothing in the CHAIN methodology which is intrinsically linked with finite-state model checking; any methodology which can give rise to proofs and which allows for the discovery of reduction/equivalence relations among sets of configurations can just as well act as the basis for defining our property of interest π and the set of reduction rules $\mathcal{R}$. As an example, in this section we show the application of this methodology to the characterization of a web cache system which employs the Basis Token Consistency protocol [4], a protocol whose correctness follows directly from the definition of vector clocks [11, 23] (its underlying conceptual mechanism).

For BTC, the interesting π is whether the client at the end of some arrangement $a \in \mathcal{A} = \mathcal{SP}^*\mathcal{C}$ will be guaranteed to see an (internally) consistent sequence of responses, *i.e.*, one which is temporally non-decreasing.[20] If $\pi(a) =$ **true**, then arrangement a will always cause the client's view of the server to be consistent (temporally non-decreasing); $\pi(a) =$ **false** indicates that arrangement a *can* provide a client with an inconsistent response. Basis Token Consistency (BTC) guarantees such consistency for any supporting cache downstream of a supporting server, regardless of the presence of intermediary inconsistent caches

[19]Another perspective on this is that all of the behavioral *modalities* of a pair of *mrouter*s will be possibilistically exhibited by a single *mrouter*, although particular *instances* of correct-drop behavior by a pair will not be exhibited by a single.

[20]Note that *recency* is not relevant in this definition. For other cache management algorithms, interesting properties to consider could be lower bounds on recency or upon hit rates under certain classes of request regimens.

$$((\mathcal{S}\ (\textit{proxy-scrubber} \mid \textit{proxy-plain})^*) \mid (\textit{server-btc}\ \mathcal{P}^*_{\textbf{clean}}\ \textit{cache-btc}\ (\textit{proxy-plain} \mid \textit{proxy-scrubber})^*) \mid (\textit{server-btc}\ \mathcal{P}^*_{\textbf{clean}}\ \textit{cache-btcpush}\ \mathcal{P}^*))\ \textit{client}$$

$$\text{Where } \mathcal{P}_{\textbf{clean}} = \{\textit{proxy-plain}, \textit{cache-plain}, \textit{cache-btc}, \textit{cache-btcpush}\}$$

Figure 8: Consistent Cache Arrangements for Theorem 4.5

(so long as intermediary proxies do not repress response headers which they do not understand). This fundamental property of BTC gives rise directly to a rewrite rule which preserves π: any number of proxies which do not "scrub" headers (*i.e.*, proxies which do not flagrantly violate the HTTP specification) between a BTC server and a BTC downstream agent (client or cache) will not affect π and can therefore be rewritten out of the set of characteristic arrangements.

A simplified model of the web for BTC's purposes uses the following agents:

$$\begin{aligned}
\mathcal{S} =& \{\textit{server-btc},\ \textit{server-plain}\},\\
\mathcal{P} =& \{\textit{proxy-scrubber},\ \textit{proxy-plain},\ \textit{cache-plain},\\
& \quad \textit{cache-btc},\ \textit{cache-btcpush}\},\\
\mathcal{C} =& \{\textit{client}\}.
\end{aligned}$$

where *cache-btcpush* uses the end-to-end strong consistency extension [3]. The inclusion of $\mathcal{C}$ is pure sugar; the interesting property as far as π is concerned is the state of the furthest downstream cache, *i.e.*, the caching agent appearing closest to the end of the arrangement. Other types of agents besides the ones described (*e.g.*, a scrubbing proxy-cache, a client with either a standard or a BTC cache, or a server implementing BTC push) can be modeled as particular sequences of these basic elements.

The definitions of standard proxying and proxy-caching in light of BTC's notion of consistency give rise to some basic reductions, such as the insertion of *proxy-plain* (cacheless proxy) agents having no effect, or indifference to the ordering of *proxy-scrubber* and *cache-plain* agents. These are reflected as reductions R_1 through R_6 in Appendix B.1. The definition and the correctness of BTC itself gives rise directly to 14 additional reductions, R_7 through R_{20} also in Appendix B.1.

These twenty rules, through the application of the CHAIN methodology, identify a sufficient subset $\mathcal{A}_{20}$ containing four member arrangements, described by the expression:

$$\mathcal{A}_\top = (\textit{server-plain}|\textit{server-btc})\ \textit{cache-plain}^{\leq 1}\ \textit{client}$$

where the two arrangements without a *cache-plain* are consistency-safe and the two containing a *cache-plain* are consistency-unsafe.

Confluence: These twenty rules are not confluent because some members of $\mathcal{A}_{\textbf{false}}$ can be rewritten to both of the false members of $\mathcal{A}_{20}$. $\mathcal{A}_{\textbf{true}}$ has a similar problem. The system can easily be made confluent, however, by adding a set of finalizing rewrites which collapse the two "true" members of $\mathcal{A}_{20}$ into one (*server-btc client* $\rhd$ *server-plain client*) and likewise for the two "false" members (*server-btc cache-plain client* $\rhd$ *server-plain cache-plain client*). The new system of twenty-two rewrite rules is confluent and produces an $\mathcal{A}_\top$ with only two member arrangements; since π is a binary property, this implies that result of the rewriting process itself maps trivially to the value of $\pi(a)$ for any $a \in \mathcal{A}$.

A π Decision Rule Using Correctness Patterns: As in Theorem 4.3, we wish to use our results to provide a computationally inexpensive rule which can decide $\pi(a)$ for any $a \in \mathcal{A}$.

Intuitively, we know that a caching system will provide the client with a consistent view under any of three circumstances: (1) there are no caches between the server (whether plain or BTC) and the client; (2) the server supports BTC, the last cache before the client is reached is a BTC cache, and there are no scrubbers between the BTC server and that final cache; (3) the server supports BTC, the system includes a *btcpush* cache, and there are no scrubbers between the server and a *btcpush* cache.

Theorem 4.5. *All caching arrangements which provide a client with a consistent view of server state (that is, all members of $\mathcal{A}_{\textbf{true}}$) will match the pattern stated in Figure 8.*

Proof. Similar to Theorem 4.3. A "safety pattern" is simply the compliment of a "failure pattern", so its validity is established by the same properties: first, whether it correctly partitions $\mathcal{A}_\top$; second, whether it defines a set which is closed under all reductions (that is, reductions preserve both membership and non-membership).

These properties are proven in Appendix B.2. □

5 Conclusion

In this paper we have presented CHAIN, an algebraic approach that enables the reduction of arbitrary arrangements of network protocol agents to a canonical, behaviorally-equivalent (with respect to some correctness property) representation, which is amenable to mechanical verification. Our methodology relies upon the discovery of a sufficient set of reduction/rewrite relationships, which establish homomorphism between particular arrangement patterns. We have applied our approach to the verification of safety properties of three examples: The HTTP request continuation protocol, an MPEG packet forwarding protocol for overlay networks, and a Web intra-cache consistency protocol.

On-going Work: In this paper, we have focused upon applying CHAIN to linear compositions of agents. For some applications, this is sufficient (*e.g.*, to model the simple case of a linear arrangement of proxies); however, such a methodology is also useful for other graph structures. For example, a particular cache may have several upstream caches available to it, any of which can be used to reach an origin server; the result is a *divergent* delivery network, in which there exists more than one path from the origin server to a cache (generally, a DAG).

CHAIN is not limited to linear composition of agents but can be applied to more compositionally rich systems, *e.g.*, trees and directed graphs. This can be accomplished by encoding sets of paths in those graphs as sets of strings and formulating the rewrite rules so as to effectively rewrite the graph by rewriting its constituent paths. We are currently developing applications of CHAIN to several graph-structured correctness problems.

Broader Vision and Research Agenda: Today, and to a large extent, "programming" distributed applications over the Internet suffers from the same lack of organizing principles as did programming of stand-alone computers some thirty years ago. Primeval programming languages were expressive but unwieldy; software engineering technology improved not only through better understanding of useful abstractions, but also by automating the process of verification of safety properties both at compile time (e.g., type checking) and run times (e.g., memory bound checks). We believe that the same kinds of improvements could find their way into the programming of distributed Internet services. The work we present in this paper is an instance of our broader goal of applying more rigorous disciplines to the specification and creation of networked protocols, programs, and services. The development of CHAIN is an important milestone in our on-going efforts to provide a sound framework for integrating a wide range of proof and verification strategies with the principles of design, development, compilation and execution of disciplined and safe programmable systems.

References

[1] Franz Baader and Tobias Nipkow. *Term* Re*writing and* All That. Cambridge University Press, 1998.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[3] Adam D. Bradley and Azer Bestavros. Basis token consistency: Extending and evaluating a novel web consistency algorithm. In *Workshop on Caching, Coherence, and Consistency (WC3)*, New York, June 2002. IEEE.

[4] Adam D. Bradley and Azer Bestavros. Basis token consistency: Supporting strong web cache consistency. In *Global Internet Worshop*, Taipei, November 2002. IEEE.

[5] Adam D. Bradley, Azer Bestavros, and Assaf J. Kfoury. Safe composition of web communication protocols for extensible edge services. In *Workshop on Web Caching and Content Delivery (WCW)*, Boulder, CO, August 2002.

[6] 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.

[7] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *POPL 2002*, Portland, OR, January 2002.

[8] Marsha Chechik, Benet Devereux, and Arie Gurfinkel. Model-checking infinite state-space systems with fine-grained abstractions using spin. In *8th SPIN Workshop*, Toronto, 2001.

[9] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification*, volume 939, pages 54–69, Liege, Belgium, 1995. Springer Verlag.

[10] Yifei Dong, Xiaoqun Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David Scott Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 74–88, 1999.

[11] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

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

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

[14] Timothy G. Griffin and Gordon Wilfong. On the correctness of IBGP configuration. In *ACM SIGCOMM*, Pittsburgh, PA, August 2002. ACM.

[15] Dan He, Gilles Muller, and Julia L. Lawall. Distributing MPEG movies over the internet using programmable networks. In *International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

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

[17] Daniel Jackson. Abstract model checking of infinite specifications. In *Proceedings of Formal Methods Europe*, Barcelona, October 1994.

[18] Donald E. Knuth and P.B. Bendix. Simple word problems in universal algebra. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[19] 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.

[20] Antonín Kučera and Petr Jančar. Equivalence-checking with infinite-state systems: Techniques and results. In *Proceedings of 29th Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM 2002)*, pages 41–73. Springer-Verlag, 2002.

[21] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99), LNCS 1817*, pages 63–82, Venice, Italy, 2000.

[22] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[23] F. Mattern. Virtual time and global states of distributed systems. In *Proc. Parallel and Distributed Algorithms Conf.*, pages 215–226, 1988.

[24] 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/.

[25] Vern Paxson. End-to-end internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[26] Scott Shenker. Where's the science? Keynote Address presented at the NSF ANIR Principle Investigators Meeting, Reston, VA, January 2003.

[27] Joe Touch, David Wagner, and Jean Walrand. Panel recommendations on "Network Trustworthiness". Presented at the NSF ANIR Principle Investigator Meeting, Reston, VA, January 2003.

A HTTP Deadlock-Safety

For these lemmas, we refer to the union of the subsets of $\mathcal{A}$ defined by each of the two patterns (*i.e.*, the subset of $\mathcal{A}$ which match either or both expressions in Figure 3) as the "pattern space".

Lemma A.1. *For all $a \in \mathcal{A}_\perp$, $\pi(a) =$ **false** iff a is in the pattern space.*

Proof. We prove this result by brute force. The members of $\mathcal{A}_\perp$ are defined by Figure 3:

$$(\text{C0 P2} \mid \text{C1 P2}^{\leq 1} \mid \text{C2})\ \text{P1}\ \ \text{P1}^{\leq 1}(\text{S2} \mid \text{P2}^{\leq 1}(\text{S0} \mid \text{S1})) \mid \mathcal{CS}$$

For each member of this set, we determine whether it matched either of the patterns; eight (8) match only the first pattern, two (2) match only the second pattern, and five (5) match both, meaning the pattern suggests that only these 15 of the 49 members of $\mathcal{A}_\perp$ are deadlock-prone.

We then compute $\pi(a)$ for all $a \in \mathcal{A}_\perp$ arrangements, and found 15 arrangements to be deadlock-prone. These fifteen are the same arrangements which were identified by the patterns above. Therefore, the intersection of the pattern space with $\mathcal{A}_\perp$ is precisely the set explicitly found to be $\mathcal{A}_\perp \cap \mathcal{A}_{\textbf{false}}$. □

Lemma A.2. *All arrangements in the pattern space can be reduced to members of $\mathcal{A}_{\textbf{false}} \cap \mathcal{A}_\perp$.*

Proof. We already have shown that all members of $\mathcal{A}$ can be reduced to members of $\mathcal{A}_\perp$. As such, if all reductions preserve membership in $\mathcal{A}_{\textbf{false}}$ (that is, if $(\pi(a) = \textbf{false}) \rightarrow (\pi(f_i^{(1)}(a)) = \textbf{false})$ for all reduction functions f_i), then all members of $\mathcal{A}_{\textbf{false}}$ can clearly be reduced to members of $\mathcal{A}_{\textbf{false}} \cap \mathcal{A}_\perp$.

We have factored the pattern space into a set of seven regular expression which will be easier to reason about.

1. $\mathcal{C}\ \mathcal{P}^*$ P1 P1 (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$)
2. $\mathcal{C}\ \mathcal{P}^*$ P1 P2$^+$ (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$)
3. C1 P1 (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$)
4. C1 P2$^+$ (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$)
5. $\mathcal{C}\ \mathcal{P}^*$ P1 (P1 | P2)* P1 (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$)
6. C1 (P1 | P2)* P1 (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$)
7. C2 (P1 | P2)* P1 (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$)

We now examine each of the 10 reduction rules and show that $(\pi(a) = \textbf{false}) \rightarrow (\pi(f_i^{(1)}(a)) = \textbf{false})$ for any arrangements which are members of the sets described by the above seven patterns.

R_1 : x P0 $y = x$ S0, C0 y

Consider an arrangement matching any of the seven patterns which contains a P0. If that P0 happens to correspond with the head of the (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$) sub-pattern (common to all patterns), then clearly at least the x S0 produced arrangement will match the same original pattern. Thus, R_1 preserves membership in the pattern space.

R_2 : x P0$^+$ $y = x$ P0 y

Consider an arrangement matching any of the seven patterns which contains one or more sequence of P0s. Those P0s are in one of two locations: either the leading $\mathcal{C}\ \mathcal{P}^*$ sub-pattern or the trailing (S0 | P0 $\mathcal{P}^*\ \mathcal{S}$) sub-pattern. In both cases, clearly the removal of some P0s will not effect the match, since all P0s beyond the first one (and in the former case, even the first one) are matched by a $\mathcal{P}^*$. Thus, R_2 preserves membership in the pattern space.

R_3 : C2 P2* $x =$ C2 x

Consider each of the seven patterns:

1,2,5 The removed P2s are matched by the $\mathcal{P}^*$ term, so clearly their removal will not effect the match.

3,4,6 Does not apply.

7 The P2s are matched by the (P1 | P2)* sub-pattern, so clearly their removal will not effect the match.

Thus, R_3 preserves membership in the pattern space.

R_4 : x P2* S2 $= x$ S2

Consider an arrangement matching any of the seven patterns which contains the subsequence P2* S2. Clearly, this subsequence can only match the (common) $\mathcal{P}^*$ $\mathcal{S}$ sub-pattern, and thus the removal of the P2s will not effect the match. Thus, R_4 preserves membership in the pattern space.

R_5 : x P2$^+$ $y = x$ P2 y

Consider each of the seven patterns:

1,3 P2s can only appear in the $\mathcal{P}^*$ terms, so their removal will not effect the match.

2,4 P2s appear in the $\mathcal{P}^*$ and P2$^+$ terms, so the removal of those beyond the first will not effect the match.

5,6,7 P2s appear in the $\mathcal{P}^*$ and $(\text{P1} \mid \text{P2})^*$ sub-patterns, so their removal will not effect the match.

Thus, R_5 preserves membership in the pattern space.

R_6 : x P1$^{\geq 2}$ $y = x$ P1 P1 y

Consider the seven patterns:

1,2 All P1s beyond the second in a sequence must match within either the $\mathcal{C}$ $\mathcal{P}^*$ sub-pattern or the $\mathcal{P}^*$ $\mathcal{S}$ sub-pattern, so their removal does not effect the match.

3,4 All P1 sequences of length greater than one must match within the $\mathcal{P}^*$ $\mathcal{S}$ sub-pattern, so their removal does not effect the match.

5 All P1 sequences of length greater than one must match within either the $\mathcal{P}^*$ P1 $(\text{P1} \mid Pc)^*$ P1 sub-pattern or the $\mathcal{P}^*$ $\mathcal{S}$ sub-pattern; in the former case, removing all beyond the second P1 will not effect the match (the sub-pattern demands at most the outer two P1s), and in the latter case, removal of P1s will not effect the match.

6,7 All P1 sequences must match either the $(\text{P1} \mid \text{P2})^*$ P1 sub-pattern or the $\mathcal{P}^*$ term; in each case, the removal of P1s beyond the second will not effect the match.

Thus, R_6 preserves membership in the pattern space.

R_7 : C0 P1 $x =$ C1 x

Consider the seven patterns:

1 If the leading C0 P1 matches the $\mathcal{C}$ $\mathcal{P}^*$ sub-pattern, then the reduction has no effect. If the leading C0 P1 matches the $\mathcal{C}$ $\mathcal{P}^*$ P1 sub-pattern, then the whole arrangement is rewritten to one of the form C1 P1 $(\text{S0} \mid \text{P0}\ \mathcal{P}^*\ \mathcal{S})$, which is precisely the set recognized by pattern 3.

2 Similar to 1 above; in the latter case, the whole arrangement is rewritten to one of the form

$$\text{C1 P2}^+\ (\text{S0} \mid \text{P0}\ \mathcal{P}^*\ \mathcal{S})$$

which is precisely the set recognized by pattern 4.

3,4,6,7 Reduction does not apply

5 Similar to 1 above; in the latter case, the whole arrangement is rewritten to have the form

$$\text{C1}\ (\text{P1} \mid \text{P2})^*\text{P1}\ (\text{S0} \mid \text{P0}\ \mathcal{P}^*\ \mathcal{S})$$

which is precisely the set recognized by pattern 6.

Thus, R_7 preserves membership in the pattern space.

R_8 : x P1 P2 P1 $y = x$ P1 P1 y

Consider the seven patterns:

1,2 The subsequence P1 P2 P1 can only match either the $\mathcal{P}^*$ P1 sub-pattern or the $\mathcal{P}^*$ term; in either case, removing the P2 will not effect the match.

3,4 The subsequence P1 P2 P1 can only match the $\mathcal{P}^*$ term, so removing the P2 will not effect the match.

5 The subsequence P1 P2 P1 can match either the $\mathcal{P}^*$ P1, P1 $(\text{P1} \mid \text{P2})^*$, or P1 $(\text{P1} \mid \text{P2})^*$ P1 sub-patterns, or one of the $\mathcal{P}^*$ terms, so in all cases the removal of the P2 will not effect the match.

6,7 The subsequence P1 P2 P1 can match the $(\text{P1} \mid \text{P2})^*$ or $(\text{P1} \mid \text{P2})^*$ P1 sub-patterns, or the $\mathcal{P}^*$ term, so in all cases the removal of the P2 will not effect the match.

Thus, R_8 preserves membership in the pattern space.

R_9 : C1 P1 P1 $x =$ C1 P1 x

1 If $\mathcal{C}$ $\mathcal{P}^* =$ C1, then the rewrite will match with pattern 3; otherwise, will not effect the match.

2 If $\mathcal{C}$ $\mathcal{P}^* =$ C1 P1, then the rewrite will match with pattern 4; otherwise, will not effect the match.

3 If it matches, the second P1 matches within $\mathcal{P}^*$, so removal will not effect the match.

4,7 Does not apply.

5 Removed P1 will either be within $\mathcal{P}^*$ or the first symbol in $(\text{P1} \mid \text{P2})$ P1, so removal will not effect the match.

6 If P1 P1 matches entirely within $(\text{P1} \mid \text{P2})^*$, removing one P1 has no effect upon matching; otherwise, matches with 3.

Thus, R_9 preserves membership in the pattern space.

R_{10} : C1 P2 P1 $x =$ C1 P1 x

1,2,5 Removed P2 will match within $\mathcal{P}^*$, so removal does not effect the match. $\mathcal{C}$ $\mathcal{P}^*$ P1 P1 (S0 | P0 $\mathcal{P}^*$ $\mathcal{S}$)

3,4,7 Does not apply.

6 Removed P2 will match within (P1 | P2)*, so removal does not effect the match. C1 (P1 | P2)* P1 (S0 | P0 $\mathcal{P}^*$ $\mathcal{S}$)

Thus, R_{10} preserves membership in the pattern space.

All reductions ($R_1 \ldots R_{10}$) preserve membership in the pattern space. □

Lemma A.3. *No arrangements outside the pattern space can be reduced to members of the pattern space.*

Proof. This proof is the compliment to that of Lemma A.2; rather than proving that reductions preserve membership, we prove that reductions also preserve non-membership.

To do this, it suffices to show that the inversions of the rewrite functions derived from our reductions can not be used to produce an arrangement which is not a member of the pattern space from one which is. From this it follows inductively that no non-member arrangement can be reduced to a member arrangement.

The inverted rewrite rules (we will call them "productions") are easily derived for the equivalences behind R_2 through R_{10}. Equivalence R_1 tells us that we can treat the two sub-terms (S0 and P0 $\mathcal{P}^*$ $\mathcal{S}$) of the common closing sub-pattern (S0 | P0 $\mathcal{P}^*$ $\mathcal{S}$) as being equivalent for the purposes of matching; thus, we do not treat the P0 $\mathcal{P}^*$ $\mathcal{S}$ sub-pattern further in this proof, since any deadlock-inducing subsequences which that sub-pattern would match will be matched by the more "substantial" parts of the patterns (the sub-pattern preceding S0).

We now examine the inversions of f_2 through f_{10} (corresponding with R_2 through R_{10}, respectively) and show that $(\pi(a) = \textbf{false}) \rightarrow ((\pi(b) = \textbf{false})\ \forall b \in f_i^{-1}(a))$. For each, we refer to the same seven-way factoring of the pattern space used to prove Lemma A.2.

f_2^{-1} : x P0 $y = x$ P0$^+$ y

In all cases, additional P0 are captured in a $\mathcal{P}^*$. Thus, f_2^{-1} preserves membership in the pattern space.

f_3^{-1} : C2 $x =$ C2 P2* x

1,2,5 Additional P2 captured by $\mathcal{C}$ $\mathcal{P}^*$.

3,4,6 Does not apply

7 Additional P2 captured by (P1 | P2)*.

Thus, f_3^{-1} preserves membership in the pattern space.

f_4^{-1} : x S2 $= x$ P2* S2

Additional P2 captured by $\mathcal{P}^*$ $\mathcal{S}$. Thus, f_4^{-1} preserves membership in the pattern space.

f_5^{-1} : x P2 $y = x$ P2$^+$ y

1,3 Additional P2 captured by $\mathcal{P}^*$.

2,4 Additional P2 captured by $\mathcal{P}^*$ and P2$^+$.

5,6,7 Additional P2 captured by $\mathcal{P}^*$ and (P1 | P2)*.

Thus, f_5^{-1} preserves membership in the pattern space.

f_6^{-1} : x P1^2 $y = x$ P1$^{\geq 2}$ y

1,2 Additional P1 captured by $\mathcal{P}^*$.

3,4 Does not apply.

5 Additional P1 captured by combination of $\mathcal{P}^*$ and (P1 | P2)*.

6,7 Additional P1 captured by (P1 | P2)* or $\mathcal{P}^*$.

Thus, f_6^{-1} preserves membership in the pattern space.

f_7^{-1} : C1 $x =$ C0 P1 x

1,2,5 Effect captured by $\mathcal{C}$ $\mathcal{P}^*$.

3 Produces arrangement which matches pattern 1.

4 Produces arrangement which matches pattern 2.

6,7 Produce arrangements which match either pattern 1 or pattern 2.

Thus, f_7^{-1} preserves membership in the pattern space.

f_8^{-1} : x P1^2 $y = x$ P1 P2 P1 y

1 Produces arrangement which matches pattern 5.

2 Additional P2 captured by $\mathcal{P}^*$.

3,4 Does not apply.

5 Additional P2 captured by either $\mathcal{P}^*$ or (P1 | P2)*.

6,7 Additional P2 captured by (P1 | P2)* or $\mathcal{P}^*$.

Thus, f_8^{-1} preserves membership in the pattern space.

f_9^{-1} : C1 P1 $x =$ C1 P1$^+$ x

1,2,5 Additional P1 captured by $\mathcal{P}^*$.

3 Produces arrangement which matches pattern 1.

4,7 Does not apply.

6 Additional P1 captured by (P1 | P1)*.

Thus, f_9^{-1} preserves membership in the pattern space.

f_{10}^{-1} : C1 P1 $x =$ C1 P2 P1 x

1,2,5 Additional P2 captured by $\mathcal{P}^*$.

3 Produces arrangement which matches pattern 6.

4,7 Does not apply.

6 Additional P2 captured by (P1 | P2)*.

Thus, f_{10}^{-1} preserves membership in the pattern space.

Thus, no arrangement which can be reduced to one within the pattern space is itself outside of the pattern space; therefore, no non-member can be reduced to a member, so non-membership is preserved by the ten reductions. □

1	(*server-plain* | *server-btc*) (*proxy-scrubber* | *proxy-plain*)* *client*
2	*server-btc* $\mathcal{P}^*_{\textbf{clean}}$ *cache-btc* (*proxy-plain* | *proxy-scrubber*)* *client*
3	*server-btc* $\mathcal{P}^*_{\textbf{clean}}$ *cache-btcpush* $\mathcal{P}^*$ *client*

Table 3: BTC π decision rule patterns

B Web Intra-Cache Consistency

B.1 Reduction Rules

R_1 : x *proxy-plain* $y \ \triangleright \ x\ y$
(plain proxying has no effect upon caching)

R_2 : x *cache-plain cache-plain* $y \ \triangleright \ x$ *cache-plain* y
(plain proxying has no incremental/marginal effect)

R_3 : x *proxy-scrubber proxy-scrubber* $y \ \triangleright$ x *proxy-scrubber* y
(adjacent scrubbers have no incremental/marginal effect)

R_4 : x *proxy-scrubber cache-plain* $y \ \triangleright$ x *cache-plain proxy-scrubber* y
(because *cache-plain*ignored BTC headers, the order of this pair doesn't matter, so we normalize it)

R_5 : *server-plain proxy-scrubber* $x \ \triangleright$ *server-plain* x
(a scrubbing proxy has no effect upon a plain server, as there is nothing to scrub)

R_6 : x *proxy-scrubber client* $\triangleright \ x$ *client*
(a scrubbing proxy has no effect upon a cacheless client)

R_7 : x *cache-btc cache-btc* $y \ \triangleright \ x$ *cache-btc* y
(successive BTC caches add no incremental value)

R_8 : x *cache-btcpush cache-btcpush* $y \ \triangleright$ x *cache-btcpush* y
(as R_7)

R_9 : x *cache-btcpush cache-btc* $y \ \triangleright \ x$ *cache-btcpush* y
(as R_7, and "push" effect passes through the *cache-btc*)

R_{10} : x *cache-btc cache-btcpush* $y \ \triangleright \ x$ *cache-btcpush* y
(as R_7, and "push" effects are only downstream)

R_{11} : x *cache-btcpush cache-plain* $y \ \triangleright$ x *cache-btcpush proxy-plain* y
("push" protocol disables downstream non-BTC caches)

R_{12} : x *proxy-scrubber cache-btc* $y \ \triangleright$ x *proxy-scrubber cache-plain* y
(no tokens reach the *cache-btc*, therefore it defaults to acting like a regular cache)

R_{13} : x *proxy-scrubber cache-btcpush* $y \ \triangleright$ x *proxy-scrubber cache-plain* y
(as R_{12})

R_{14} : x *cache-plain cache-btc* $y \ \triangleright \ x$ *cache-btc* y
(a BTC cache is unaffected by inconsistency introduced by the upstream *cache-plain*; if it receives tokens, it acts like a correct BTC cache, otherwise it behaves like a *cache-plain*.)

R_{15} : x *cache-plain cache-btcpush* $y \triangleright x$ *cache-btcpush* y
(as R_{14})

R_{16} : *server-plain cache-btc* $x \ \triangleright$ *server-plain cache-plain* x
(a plain server provides no BTC annotations, so *cache-btc* reverts to *cache-plain* behavior)

R_{17} : *server-plain cache-btcpush* $x \ \triangleright$ *server-plain cache-plain* x
(as R_{16})

R_{18} : *server-btc cache-btc* $x \ \triangleright$ *server-btc* x
(trivially follows from the correctness of BTC)

R_{19} : x *cache-btcpush client* $\triangleright \ x$ *cache-btc client*
(no agents separate the *cache-btcpush* from the *client*, so its "push" component has no effect)

R_{20} : *server-btc proxy-scrubber* $x \ \triangleright$ *server-plain* x
(an immediately-scrubbed BTC server is indistinguishable from a plain server)

B.2 Proof of Closure under Reductions

Lemma B.1. *The union of the patterns given in Theorem 4.5 correctly identify all members of* $\mathcal{A}_\perp \cap \mathcal{A}_{\textbf{true}}$.

Proof. The "brute-force" proof of this lemma is trivial, as $\mathcal{A}_{22}$ has two members: *server-plain client* and *server-plain cache-plain client*. Among these, *server-plain client* matches the first pattern which should place it within $\mathcal{A}_{\textbf{true}}$; this agrees with a trivial analysis (there are no caches to introduce inconsistencies in either). Similarly, *server-plain cache-plain client* does not match any patterns, identifying it as a member of $\mathcal{A}_{\textbf{false}}$; this agrees with a trivial analysis (there is nothing to prevent the *cache-plain* agent from introducing inconsistencies, as "plain" caches are able to do by their nature). So the pattern has correctly partitioned $\mathcal{A}_\perp$. □

Lemma B.2. *The set defined by the union of the three patterns given in Theorem 4.5 is closed under all equivalence rules;* i.e., *for all* f_i *and* f_i^{-1} *corresponding with valid* R_i, *it is true for all* $a \in \mathcal{A}$ *that* $\pi(a) = \pi(f_i^*(a))$ *and that* $\pi(a) = \pi(f_i^{-1(*)})(a)$.

Proof. For each of the 22 reductions R_1 through R_{22}, we examine its meaning with respect to members of each of the three success patterns stated in Theorem 4.5 (re-stated in Table 3.

The 22 reductions are stated below as equivalences, with discussions of their effects (both as reductions and as productions) upon each of the three success patterns. "Does not apply" indicates that neither side of the equivalence corresponds with members of the pattern, which implies that the equivalence preserves non-membership.

R_1 : x *proxy-plain* $y \;=\; x\; y$

1 Effected *proxy-plain* would appear in/be removed from $($*proxy-scrubber* $\mid$ *proxy-plain*$)^*$ sub-pattern; no effect upon match.

2 Effected *proxy-plain* would appear in/be removed from $\mathcal{P}^*_{\textbf{clean}}$ term and $($*proxy-plain* $\mid$ *proxy-scrubber*$)^*$ sub-pattern; no effect upon match.

3 Effected *proxy-plain* would appear in/be removed from $\mathcal{P}^*_{\textbf{clean}}$ and $\mathcal{P}^*$ terms; no effect upon match.

Thus, R_1 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_2 : x *cache-plain cache-plain* $y \;=\; x$ *cache-plain* y

1 Does not apply.

2 Effected *cache-plain* would appear in/be removed from $\mathcal{P}^*_{\textbf{clean}}$ term; no effect upon match.

3 As under R_1.

Thus, R_2 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_3 : x *proxy-scrubber proxy-scrubber* $y \;=\; x$ *proxy-scrubber* y

1 As under R_1.

2 Effected *proxy-scrubber* would appear in/be removed from $($*proxy-plain* $\mid$ *proxy-scrubber*$)^*$ sub-pattern; no effect upon match.

3 Effected *proxy-scrubber* would appear in/be removed from $\mathcal{P}^*$ term; no effect upon match.

Thus, R_3 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_4 : x *proxy-scrubber cache-plain* $y \;=\; x$ *cache-plain proxy-scrubber* y

1,2 Does not apply.

3 As under R_3.

Thus, R_4 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_5 : *server-plain proxy-scrubber* $x \;=\;$ *server-plain* x

1 Effected *proxy-scrubber* would appear in/be removed from $($*proxy-scrubber* $\mid$ *proxy-plain*$)^*$ sub-pattern; no effect upon match.

2,3 Does not apply.

Thus, R_5 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_6 : x *proxy-scrubber client* $\;=\; x$ *client*

1,2 As under R_5.

3 Effected *proxy-scrubber* would appear in/be removed from $\mathcal{P}^*$ term; no effect upon match.

Thus, R_6 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_7 : x *cache-btc cache-btc* $y \;=\; x$ *cache-btc* y

1 Does not apply.

2 Effected *cache-btc* would appear in/be removed from $\mathcal{P}^*_{\textbf{clean}}$ term; no effect upon match.

3 Effected *cache-btc* would appear in/be removed from $\mathcal{P}^*_{\textbf{clean}}$ and $\mathcal{P}^*$ terms; no effect upon match.

Thus, R_7 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_8 : x *cache-btcpush cache-btcpush* $y \;=\; x$ *cache-btcpush* y

1 Does not apply.

2,3 As under R_7, but for *cache-btcpush*.

Thus, R_8 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_9 : x *cache-btcpush cache-btc* $y \;=\; x$ *cache-btcpush* y

1 Does not apply.

2,3 As under R_7.

Thus, R_9 and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{10} : x *cache-btc cache-btcpush* $y \;=\; x$ *cache-btcpush* y

1 Does not apply.

2,3 As under R_8.

Thus, R_{10} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{11} : x *cache-btcpush cache-plain* $y \;=\; x$ *cache-btcpush proxy-plain* y

1 Does not apply.

2 Effected *cache-plain* or *proxy-plain* would be substituted within $\mathcal{P}^*_{\textbf{clean}}$ term; no effect upon match.

3 Effected *cache-btc* or *proxy-plain* would be substituted within $\mathcal{P}^*_{\textbf{clean}}$ or $\mathcal{P}^*$ terms; no effect upon match.

Thus, R_{11} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{12} : x *proxy-scrubber cache-btc* y $=$ x *proxy-scrubber cache-plain* y

1,2 Does not apply.

3 Effected *cache-btc* or *cache-plain* would be substituted within $\mathcal{P}^*$ term; no effect upon match.

Thus, R_{12} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{13} : x *proxy-scrubber cache-btcpush* y $=$ x *proxy-scrubber cache-plain* y

1,2 Does not apply.

3 As under R_{12}, but for *cache-btcpush* or *cache-plain.*

Thus, R_{13} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{14} : x *cache-plain cache-btc* y $=$ x *cache-btc* y

1 Does not apply.

2 Effected *cache-plain* would appear in/be removed from $\mathcal{P}^*_{\textbf{clean}}$ term; no effect upon match.

3 Effected *cache-plain* would appear in/be removed from $\mathcal{P}^*_{\textbf{clean}}$ or $\mathcal{P}^*$ terms; no effect upon match.

Thus, R_{14} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{15} : x *cache-plain cache-btcpush* y $=$ x *cache-btcpush* y

1 Does not apply.

2,3 As under R_{14}, but for *cache-btcpush.*

Thus, R_{15} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{16} : *server-plain cache-btc* x $=$ *server-plain cache-plain* x

1,2,3 Does not apply.

Thus, R_{16} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{17} : *server-plain cache-btcpush* x $=$ *server-plain cache-plain* x

1,2,3 Does not apply.

Thus, R_{17} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{18} : *server-btc cache-btc* x $=$ *server-btc* x

1 Reduction does not apply; production results in an arrangement matching pattern 2.

2 For reduction, effected *cache-btc* may be removed from $\mathcal{P}^*_{\textbf{clean}}$ (no effect upon match), or could be the explicit *cache-btc* in which case the result is an arrangement matching pattern 1. For production, the effected *cache-btc* would appear in $\mathcal{P}^*_{\textbf{clean}}$ (no effect upon match).

3 Effected *cache-btc* would appear in/be removed from $\mathcal{P}^*_{\textbf{clean}}$ term; no effect upon match

Thus, R_{18} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{19} : x *cache-btcpush client* $=$ x *cache-btc client*

1 Does not apply

2 Left-to-right production does not apply; right-to-left production results in an arrangement matching pattern 3

3 Left-to-right production results in an arrangement matching pattern 2; for right-to-left production, effected *cache-btc* and *cache-btcpush* would be substituted within $\mathcal{P}^*$ term (no effect upon match).

Thus, R_{19} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{20} : *server-btc proxy-scrubber* x $=$ *server-plain* x

1 Effected *proxy-scrubber* would appear in/be removed from $($*proxy-scrubber* $|$ *proxy-plain*$)^*$ sub-pattern; no effect upon match.

2,3 Does not apply.

Thus, R_{20} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{21} : *server-btc client* $=$ *server-plain client*

1 Both sides of this rewrite match this pattern.

2,3 Does not apply.

Thus, R_{21} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

R_{22} : *server-btc cache-plain client* $=$ *server-plain cache-plain client*

1,2,3 Does not apply.

Thus, R_{22} and its inverse both preserve membership in $\mathcal{A}_{\textbf{true}}$.

Thus, the sets $\mathcal{A}_{\textbf{false}}$ and $\mathcal{A}_{\textbf{true}}$ are each closed under the reductions R_1 through R_{22}. □