Validating Arbitrarily Large Network Protocol Compositions with Finite Computation [*]

Adam D. Bradley, Azer Bestavros, and Assaf J. Kfoury

artdodge,best,kfoury@cs.bu.edu

Computer Science Department

Boston University

Boston, MA 02215

BUCS-TR-2002-030

November 8, 2002

Abstract

Formal tools like finite-state model checkers have proven useful in verifying the correctness of systems of bounded size and for hardening single system components against arbitrary inputs. However, conventional applications of these techniques are not well suited to characterizing emergent behaviors of large compositions of processes. In this paper, we present a methodology by which arbitrarily large compositions of components can, if sufficient conditions are proven concerning properties of small compositions, be modeled and completely verified by performing formal verifications upon only a finite set of compositions. The sufficient conditions take the form of reductions, which are claims that particular sequences of components will be causally indistinguishable from other shorter sequences of components. We show how this methodology can be applied to a variety of network protocol applications, including two features of the HTTP protocol, a simple active networking applet, and a proposed web cache consistency algorithm. We also doing discuss its applicability to framing protocol design goals and to representing systems which employ non-model-checking verification methodologies. Finally, we briefly discuss how we hope to broaden this methodology to more general topological compositions of network applications.

Keywords: Protocol Verification, Formal Methods, Model Checking, Language Reduction, Protocol Design

[*]This research was supported in part by NSF (awards ANI-0205294, ANI-9986397, ANI-0095988, CCR-9988529, and ITR-0113193) and U.S. Department of Education (GAANN Fellowship)

1 Introduction

Creating new protocols, programs, and services for the Internet currently suffers from the same lack of organizing principles as did programming of stand-alone computers some thirty years ago. While primeval programming languages were expressive, they were also unwieldy and difficult to reason about. Programming language technology improved through better understanding of useful abstraction mechanisms for controlling computational processes; we would like to see the same kinds of improvements find their way into the programming of distributed Internet services.

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

In this paper, we focus upon the problem of verifying correctness of compositions of processes of such a nature that the composition can be iteratively applied arbitrarily many times. In particular, we present a methodology which allows us to describe the correctness (or some other property) of arbitrarily large compositions of processes when certain sufficient properties can be proven concerning local interactions among small compositions of such processes.

1.1 Finite-State Model Checking

Finite-state model checking [9] is proving to be a powerful tool for assessing the correctness of certain classes of protocols [16, 15, 6] and programs [19, 17, 20, 10]. Systems like SPIN have proven particularly powerful in instances where the problem at hand demands modeling of a fixed number of interacting processes, or in which a single process must be tested against the set of all possible input sequences.

However, certain classes of protocol errors rely upon particulars of multi-process coordination and interactions which may only emerge for certain particular compositions of non-trivial numbers of agents. For example, in modeling the HTTP protocol's `100 Continue` feature [6], we uncovered both well-known and previously undocumented conditions under which HTTP/1.1 proxies could induce deadlocks in the course of conducting a continuation-expectant transaction. These deadlocks arise as a consequence of the interaction between particular processes' sequences of send and receive operations and the presence and absence of mandatory timeout conditions. As a result, detecting these failure conditions requires explicitly testing particular sequences of agents, a problem which becomes intractable because the number of agents which can be composed to form a valid system is (at least hypothetically) unbounded.

However, in our work we also manually deduced that when certain processes were composed in particular patterns, they become equivalent (from an input-output causality standpoint) to particular single processes models or other compositions of processes. We then postulated that, given enough sufficiently powerful such *reduction rules*, the infinite set of possible compositions of processes could be fully characterized by modeling a finite subset and defining a mapping from all possible compositions to members of that subset.

This paper presents the generalized methodology postulated in our previous work, showing how such reduction and equivalence relations can be applied to an infinite language of testing candidates to yield a finite subset language from which the original's full characterization can be easily derived. This contributes directly to our research agenda of formalizing the manner in which arbitrary and arbitrarily large compositions of systems will behave in light of particular rigorously-defined properties.

1.2 Related Work

Model checking is now a widely used technique in the areas of low-level protocol design, hardware design, and (to some extent) software verification. Essentially, a process or protocol is modeled explicitly as the product of a control-flow automaton and the space of possible variable values, producing a finite-state automaton. All possible executions of that automaton are then explored using a depth-first search. Interleavings of the executions of a set of such automata, communicating using global variables and message queues, can similarly be explored. In the course of these explorations, the model-checker verifies that certain properties are not violated; for example, the model checker may verify that the system never enters a deadlocked state (where all processes are waiting for external events), or that some claim expressed using a temporal logic is never violated.

Explicit modeling of all the interleavings of steps among any particular set of intercommunicating processes is a computationally very expensive operation. Fortunately, a class of optimizations based upon partial-order techniques [13, 18] can significantly decrease the necessary complexity by taking advantage of the fact that only a few state transitions represent model events which actually affect changes in the state of communication channels and other processes. This allows the model checker to omit verification of most interleavings of purely "local" transitions.

The technique of testing a single agent (automata) against a *maximal automaton* is also widely used [15] to ensure that a single agent or system is robust in the face of all possible inputs and sequences of inputs. While this is useful while trying to guarantee a localizable property of a system, it may not be sufficient for proving emergent global properties which depend upon particular classes of valid inputs and blocking operations (such as the deadlock conditions which can emerge in HTTP proxy systems).

In this paper, we rely heavily upon standard theoretical constructs such as finite-state automata, regular expressions, and the mappings between them. We also rely upon the concept of two systems of process models being "causally equivalent" as the basis for what we call "reductions"; a number of approaches exist by which this property could be established, the best choice depending upon the application at hand. A few examples are finite-state-machine equivalence/inequivalence, applications of type theory to the π-calculus [8], and the concept of "implementation" from I/O automata theory [22, 23].

1.3 Outline

We will begin in Section 2 by defining a set of theoretical formalisms which describe communication networks, linear compositions of processes, and meaningful reductions upon an infinite test space.

Section 3 expounds upon four examples which put our framework to use. In the first, we apply our formalism verbosely and illustratively to the deadlock-safety of the HTTP/1.1 Expect/Continue feature, significantly expanding upon our work in [6]. Second, we examine a similar application of our methodology to HTTP persistent connections.

In our third example, we explore issues of correctness with respect to an MPEG-oriented active networking applet. In so doing, we broaden the utility of our framework in two ways: first, by broadening our notion of what can constitute an "agent" in the context of our formalism, and second by discussing its value as a specification framework rather than an analysis framework.

Fourth, we turn our attention to the issue of web intra-cache consistency (as defined in [5]) to illustrate the independence of our framework from any particular proof strategy. This application also illustrates a system model which our methodology cannot capture, which is discussed along with other intended future research questions and concluding thoughts in Section 4.

2 Formalisms

In this section we present a set of theoretical constructs which capture the essential properties of our methodology. In the interest of brevity we have omitted some generality from this presentation; some points where our formalism is easily generalized are commented on briefly in the text or in footnotes.

Our presentation of these formalisms is clarified by our inclusion of an application well-suited to our methodology: assessing the deadlock-safety of HTTP's expect/continue feature. It has been known for some time that the `100 Continue` feature as it was originally presented in RFC2068 (the first public specification of HTTP/1.1) was ambiguous and could give rise to deadlocks between well-meaning HTTP agents; it was not clear whether the revised specification in RFC2616 had addressed all such problems, nor whether agents conforming to RFC2616 could interoperate gracefully with agents conforming to the older version without contributing to or themselves inducing other deadlock conditions. In a previous paper [6], we applied finite-state modeling to this problem and showed that certain particular sequences of agents were deadlock-prone; we were, however, unable to argue about arbitrarily long sequences of proxies connecting clients to servers. We therefore apply our methodology to this problem in order to make strong claims about all possible compositions of a client, zero or more proxies, and a server.

2.1 Communication Networks

A directed graph G is denoted by a pair $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the (non-empty) set of *nodes* and $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is the set of *edges*. A *communication network* (CN) in this paper refers to a finite directed graph $G = (\mathcal{N}, \mathcal{E})$ together with a partitioning of $\mathcal{N}$ into $m \geqslant 1$ non-empty sets:

- $\mathcal{N} = \mathcal{N}_1 \cup \cdots \cup \mathcal{N}_m$,
- $\mathcal{N}_i \neq \varnothing$ for every $1 \leqslant i \leqslant m$, and
- $\mathcal{N}_i \cap \mathcal{N}_j = \varnothing$ for $1 \leqslant i \neq j \leqslant m$.

We think of $\mathcal{N}$ as a finite set of *agents*, partitioned into $m \geqslant 1$ disjoint set of *roles*. Roles are defined by equivalence of topological structure; any member of any $\mathcal{N}_i$ can be replaced by another member of $\mathcal{N}_i$ without altering the topology of the graph.

Conceptually, the CN construct serves two distinct purposes in this paper, so we use two different monikers depending upon its intended meaning. The first is a *concrete communication network* (CCN); this represents the actual logical layout of a network, encoding each individual party to the whole communication as a node and every "physical" communication channel as a directed edge; for the purposes of our examples, it is an acyclic graph, although it need not be so in general. The second is an *abstract communication network* (ACN), which is a graph (often cyclic) where every valid path[1] represents a CCN.

As our motivating example, consider the HTTP protocol where every agent $N \in \mathcal{N}$ plays one of three roles: *client*, *proxy*, or *server*. This gives us a partitioning of $\mathcal{N}$, which we express as $\mathcal{N} = \mathcal{C} \cup \mathcal{P} \cup \mathcal{S}$. These three roles are defined topologically; for a valid ACN $G = (\mathcal{N}, \mathcal{E})$:

- G is connected.
- $\mathcal{C}$ is exactly the subset of all $N \in \mathcal{N}$ which are "sources", *i.e.*, $\mathsf{in\text{-}degree}(N) = 0$.
- $\mathcal{S}$ is exactly the subset of all $N \in \mathcal{N}$ which are "sinks", *i.e.*, $\mathsf{out\text{-}degree}(N) = 0$.
- $\mathcal{P}$ is exactly the subset of all $N \in \mathcal{N}$ which are neither "sources" nor "sinks", *i.e.*, $\mathcal{P} = \mathcal{N} - (\mathcal{C} \cup \mathcal{S})$.

This is not the only possible ACN representation; it is merely sufficient and appropriate for our example problem.

Each role is populated by *implementations*; for our purposes, this means that in the role of *client* there are three implementations as reflected by the Promela models in [6]: one implementation of the client role as specified in RFC1945 [3] (called C1945), one implementation of the client role as specified in RFC2068 [11] (called C2068), and one implementation of the client role as specified in RFC2616 [12] (called C2616). So $\mathcal{C}$, for our models, consists of all nodes which are either *client-1945*, *client-2068*, or *client-2616*. The server ($\mathcal{S}$) and proxy ($\mathcal{P}$) roles are similarly defined as $\{$P1945, P2068[2], P2616 $\}$ and $\{$S1945, S2068, S2616 $\}$, respectively.

2.2 Arrangements

Let $G = (\mathcal{N}, \mathcal{E})$ be some ACN. The set of *arrangements* in G is simply the set of finite paths in G, which we denote $\mathsf{paths}(G)$. Thus an arrangement a in G can be specified as follows: $a = N_1 N_2 \cdots N_\ell$ where $N_i \in \mathcal{N}$ for every

[1]It is interesting for many applications to consider not only linear arrangements, but other path-composing structures (*e.g.*, all valid path trees). To do so would represent a significant jump in the power of our methodology; this is discussed briefly in Section 4.1.

[2]Since *proxy-2068-hybrid* represents the most general RFC2068 proxy implementation (modeling both the *-e2e* and *-hbh* implementations as well as behaviors neither can reach), we only concern ourselves with that model for the purposes of this paper. There is no reason this methodology could not be applied to the complete set of models presented in [6], however.

$1 \leqslant i \leqslant \ell$ such that $(N_i, N_{i+1}) \in \mathcal{E}$ for every $1 \leqslant i < \ell$. Note that an arrangement a (or path in G) may have length = 0, and therefore be the sequence $a = N$ with a single entry $N \in \mathcal{N}$, rather than the empty sequence.

Depending on the desired analysis of the communication network, we consider different sets of arrangements in G. For any such analysis, we will distinguish a particular subset $\mathcal{A} \subseteq \mathtt{paths}(G)$; we call $\mathcal{A}$ the set of *valid arrangements* in G.

For example, consider our ACN $G = (\mathcal{N}, \mathcal{E})$ for the HTTP world, with a 3-part partition of $\mathcal{N} = \mathcal{C} \cup \mathcal{P} \cup \mathcal{S}$. In this case, we take $\mathcal{E}$ as the set:

$$\begin{aligned}
\mathcal{E} \;=\; & (\mathcal{P} \times \mathcal{P}) \cup && \text{all edges from } \mathcal{P} \text{ to } \mathcal{P}, \textit{ i.e.}, \text{ the edges of the complete graph over } \mathcal{P}, \\
& (\mathcal{C} \times \mathcal{P}) \cup && \text{all edges from } \mathcal{C} \text{ to } \mathcal{P}, \\
& (\mathcal{P} \times \mathcal{S}) \cup && \text{all edges from } \mathcal{P} \text{ to } \mathcal{S}, \\
& (\mathcal{C} \times \mathcal{S}) && \text{all edges from } \mathcal{C} \text{ to } \mathcal{S}.
\end{aligned}$$

Arrangements of particular interest for the HTTP world are of the form $C\, P_1 \cdots P_k\, S$ where $C \in \mathcal{C}$, $P_i \in \mathcal{P}$ and $S \in \mathcal{S}$, with $k \geqslant 0$. An arrangement of this form connects a client agent C to a server agent S using k intermediary proxy agents. These are the valid arrangements for the HTTP world. Using regular expressions, the space of all valid arrangements in this case is $\mathcal{A} = \mathcal{C}\mathcal{P}^*\mathcal{S}$. Put differently, the valid arrangements of the HTTP world are the maximal-length members of $\mathtt{paths}(G)$.

2.3 Properties

We are interested in valid arrangements in an ACN $G = (\mathcal{N}, \mathcal{E})$ that satisfy various desirable communication properties, *e.g.*, valid arrangements that are *deadlock-free*; for our purposes, it suffices to deal with a single such property at a time. Let π denote such a property, which can be viewed as a boolean-valued function $\pi : \mathcal{A} \to \{\mathbf{true}, \mathbf{false}\}$. Our first methodological goal is to obtain a "friendly" specification of the two sets:

$$\mathcal{A}_{\mathbf{true}} = \{a \in \mathcal{A} \mid \pi(a) = \mathbf{true}\} \quad \text{and} \quad \mathcal{A}_{\mathbf{false}} = \{a \in \mathcal{A} \mid \pi(a) = \mathbf{false}\}.$$

By "friendly" we mean, at a minimum, there is a feasible computation to determine whether $a \in \mathcal{A}_{\mathbf{true}}$ or $a \in \mathcal{A}_{\mathbf{false}}$; ideally, we would also like to devise an easy-to-understand formalism to describe $\mathcal{A}_{\mathbf{true}}$ and $\mathcal{A}_{\mathbf{false}}$, which can be used to quickly test whether $a \in \mathcal{A}_{\mathbf{true}}$ or $a \in \mathcal{A}_{\mathbf{false}}$. Examples of such tests, verified using the tools of this methodology, are described below in Sections 3.1 and 3.4.

This notion of a "friendly" specification is illustrated and further clarified by the analysis of our running example of the HTTP world. Henceforth, π refers to a fixed communication property; in the case of the HTTP world, we take π to be the property that a valid arrangement $a \in \mathcal{A}$ is deadlock-free with respect to the `100 Continue` behavior [6].

2.4 Reductions

Consider an arbitrary ACN $G = (\mathcal{N}, \mathcal{E})$ and a property π on the set $\mathcal{A}$ of valid arrangements in G. We denote the powerset of a set S by 2^S. We extend $\pi : \mathcal{A} \to \{\textbf{true}, \textbf{false}\}$ to a function $\pi : 2^{\mathcal{A}} \to \{\textbf{true}, \textbf{false}\}$ by defining[3] 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 a subset, not necessarily proper, of the set $\mathcal{A}$ of valid 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 $f : \mathsf{paths}(G) \to 2^{\mathsf{paths}(G)}$ 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 $\mathcal{A}'$: $\big(\bigcup_{a \in \mathcal{A}'} f(a)\big) \subsetneq \mathcal{A}'$.

We can extend $f : \mathsf{paths}(G) \to 2^{\mathsf{paths}(G)}$ to a function $f : 2^{\mathsf{paths}(G)} \to 2^{\mathsf{paths}(G)}$ 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 need $\mathcal{A}' - f(\mathcal{A}')$ to be an infinite set, *i.e.*, infinitely many valid arrangements are excluded from the search space $\mathcal{A}'$.

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

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

induced by a sequence of appropriately defined functions $f_1, f_2, \ldots, f_n$ where $f_i : \mathsf{paths}(G) \to 2^{\mathsf{paths}(G)}$ is a reduction function on $\mathcal{A}_{i-1}$ and $\mathcal{A}_i = f_i(\mathcal{A}_{i-1})$ for every $1 \leqslant i \leqslant n$. If successful, this strategy produces a finite search space $\mathcal{A}_n$ such that

$$\mathcal{A}_n = f_n(\cdots(f_2(f_1(\mathcal{A})))\cdots) \tag{1}$$

[3]Note that other formulations of $\pi(A)$ can just as well be chosen, depending upon the semantics of π; for example, $\pi(A)$ may be the logical *OR* rather than the logical *AND* of all $\pi(a)$. A partial function variation could also be used, in which $\perp$ is a result value designating an uncertain or undefined result.

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

$$\mathcal{A}_n \supseteq f_n(\cdots(f_2(f_1(a)))\cdots) \tag{2}$$

and

$$\pi(a) = \pi(f_n(\cdots(f_2(f_1(a)))\cdots)). \tag{3}$$

2.5 A Practical Approach to the 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 our two methodological goals, this one and the one mentioned in Section 2.3, by borrowing ideas from the theory of term-rewriting and by using standard techniques for manipulating regular expressions.

Consider some ACN $G = (\mathcal{N}, \mathcal{E})$. 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 $a \in \mathsf{paths}(G)$:

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

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

Consider now the set $\mathcal{A}$ of valid arrangements in G, a property π on $\mathcal{A}$, and a subset $\mathcal{A}' \subseteq \mathcal{A}$. We say that the rewrite rule R is a *reduction on* $\mathcal{A}'$ provided 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 2.4: 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 *length-decreasing* iff for all $a, b_1, \ldots, b_n \in \mathsf{paths}(G)$ such that $a \rhd_R \{b_1, \ldots, b_n\}$ it is the case that

$$\mathsf{length}(a) > \mathsf{length}(b_1)\ ,\ \ldots\ ,\ \mathsf{length}(a) > \mathsf{length}(b_n)$$

Lemma 2.1. *If the rewrite rule R is length-decreasing, then, for every $a \in \mathsf{paths}(G)$, it holds that $f_R^{(*)}(a)$ is defined, as a non-empty finite subset of* $\mathsf{paths}(G)$.

2.6 Sufficient Subspaces

The above function $f_R^{(*)}$, if its associated rewrite rule R is length-decreasing, when applied 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*.

Recall our stated strategy from Section 2.4: 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 $1 \leqslant i \leqslant n$, the function f_i will be $f_{R_i}^{(*)}$ induced by some appropriate length-decreasing rewrite rule R_i.

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 $f_R^{(*)}$ induced by R in the obvious way; for example, we say "$f_R^{(*)}$ is length-decreasing" if R is length-decreasing. It is convenient to introduce the notion of the *support* of the function $f_R^{(*)}$, or also of its associated rewrite rule R:

$$\mathsf{support}(f_R^{(*)}) \;=\; \mathsf{support}(R) \;=\; \{a \in \mathcal{A} \mid f_R^{(*)}(a) \neq \{a\}\}, \tag{4}$$

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

Lemma 2.2. *Consider a set of reductions $\mathcal{R} = \{R_1, \ldots, R_{n-1}\}$ inducing functions $\mathcal{F} = \{f_{R_1}^{(*)}, \ldots, f_{R_{n-1}}^{(*)}\}$ which are all length-decreasing. A sufficient subspace $\mathcal{A}_n$ can be defined inductively where $0 < n$ and $\mathcal{A}_0 = \mathcal{A}$ as:*

$$\mathcal{A}_n \;= \mathcal{A}_{n-1} \bigcap \left(\left(\mathcal{A} - \mathsf{support}(f_{R_i}^{(*)}) \right) \cup f_{R_i}^{(*)}(\mathsf{support}(f_{R_i}^{(*)})) \right)$$

or can be given directly by:

$$\mathcal{A}_n \;=\; \bigcap_{i=1}^{n-1} \left(\mathcal{A} - \mathsf{support}(f_{R_i}^{(*)}) \right) \cup f_{R_i}^{(*)}(\mathsf{support}(f_{R_i}^{(*)}))\;. \tag{5}$$

3 Applications

We will begin this section by completing our motivating example of HTTP Expect/Continue mechanism's deadlock-safety and showing a finite characterization of the infinite space $\mathcal{A}$. In so doing, we verbosely illustrate how the formalisms we have developed map to the problem space and its eventual solution. This is followed by a brief summary of a very similar application of our methodology to explore HTTP persistent connections.

Next, we turn our attention to different scenarios in which our framework proves useful in various ways. Section 3.3 applies our methodology to the exploration of network-layer phenomena and an example of an "active networking" applet oriented toward optimizing network resource consumption of MPEG video streams; in so doing, we discuss the concept of "reductions" as not only an artifact of specifications or implementations, but as a kind of meta-specification or constraint/design criteria for protocol specification. We then explore several alternative algorithms for the application at hand and discuss how they mesh with a set of proposed reducibility constraints.

Finally, Section 3.4 explore the Basis Token Consistency (BTC) web cache consistency algorithm [4, 5] as a system in which reductions and correctness are established via a more straightforward proof methodology, and show that our methodological analysis of BTC agrees with higher-level proofs and intuitive conclusions concerning its behavior. However, analysis of the general set of possible network environment of the BTC algorithm proves to be beyond the scope of this methodology; this shortcoming is discussed in the context of future work in Section 4.1.

3.1 HTTP Deadlock-Safety

We continue to use the example of [6] to motivate our methodology. $\mathcal{A}$ is the language of arrangements of HTTP agents: $\mathcal{A} = \mathcal{C}\mathcal{P}^*\mathcal{S}$ where

$$\begin{aligned} \mathcal{C} &= \{\ \text{C1945}, \text{C2068}, \text{C2616}\ \}, \\ \mathcal{P} &= \{\ \text{P1945}, \text{P2068}, \text{P2616}\ \}, \text{ and} \\ \mathcal{S} &= \{\ \text{S1945}, \text{S2068}, \text{S2616}\ \}, \end{aligned}$$

and π will be the deadlock-free property of an arrangement (with respect to the HTTP expectation/continuation feature); $\pi(a) =$ **true** indicates that a is not deadlock-prone, and $\pi(a) =$ **false** indicates that a is deadlock-prone. We can compute $\pi(a)$ directly for all $a \in \mathcal{A}$ using finite-state model checking; however, the time and space required to calculate $\pi(a)$ explicitly grows polynomially at best and exponentially in general with the length of a, which motivates our use of this indirect approach.

In our previous paper we proposed a set of rewrite rules (called "reductions" in that context); Table 1 presents these roughly as they were there presented, and then translated into rewrite rules as we have defined them in Section 2.5. As above, x, y, and z denote members of Var.

	Equivalence/Reduction Rule ([6])	Rewrite Rule
R_1	x P1945 y = x S1945 && C1945 y	x P1945 y $\triangleright_{R_1}$ { x S1945, C1945 y }
R_2	x P1945$^+$ y = x P1945 y && C1945 S1945	x P1945 P1945 y $\triangleright_{R_2}$ { x P1945 y, C1945 S1945 }
R_3	C2616 P2616* x = C2616 x	C2616 P2616 x $\triangleright_{R_3}$ { C2616 x }
R_4	x P2616* S2616= x S2616	x P2616 S2616$\triangleright_{R_4}$ { x S2616 }
R_5	x P2616$^+$ y = x P2616 y	x P2616 P2616 y $\triangleright_{R_5}$ { x P2616 y }
R_6	x P2068$^{\geq 2}$ y = x P2068 P2068 y	x P2068 P2068 P2068 y $\triangleright_{R_6}$ { x P2068 P2068 y }
R_7	C1945 P2068 x = C2068 x	C1945 P2068 x $\triangleright_{R_7}$ { C2068 x }

Table 1: Translating Equivalences into Rewrite Rules

All of these are proper rewrite rules as defined in Section 2.5, satisfying the three necessary conditions: 1) each has a non-empty right-hand side; 2) all strings are members of Σ^+; 3) all $\mathsf{Var}(Y_i) \subseteq \mathsf{Var}(X)$. Notice that these rules are also all length-decreasing, which implies by Lemma 2.1 that $f^{(*)}_{R_i}$ is defined. Therefore, each of the preceding rewrite rules R_i gives rise to a reduction function $f^{(*)}_{R_i}$ which is henceforth denoted f_i. The support of f_i, and the set f_i maps its support to, are presented in Table 2 using regular expressions.

	$\mathsf{support}(f_i)$	$f_i(\mathsf{support}(f_i))$
f_1	$\mathcal{C}\mathcal{P}^*\text{P1945}\mathcal{P}^*\mathcal{S}$	$\mathcal{C}(\text{P2068}\mid\text{P2616})^*\mathcal{S}$
f_2	$\mathcal{C}\mathcal{P}^*\text{P1945}\mathcal{P}^*\mathcal{S}$	$\mathcal{C}((\text{P2068}\mid\text{P2616})^*\text{P1945})^{\leq 1}((\text{P2068}\mid\text{P2616})^+\text{P1945})^*(\text{P2068}\mid\text{P2616})^*\mathcal{S}$
f_3	C2616 P2616$^+\mathcal{P}^*\mathcal{S}$	$\text{C2616}((\text{P1945}\mid\text{P2068})^+\mathcal{P}^*)^{\leq 1}\mathcal{S}$
f_4	$\mathcal{C}\mathcal{P}^*$P2616$^+$ S2616	$\mathcal{C}(\mathcal{P}^*(\text{P1945}\mid\text{P2068})^+)^{\leq 1}\text{S2616}$
f_5	$\mathcal{C}\mathcal{P}^*$ P2616$^+$ $\mathcal{P}^*\mathcal{S}$	$\mathcal{C}((\text{P1945}\mid\text{P2068})^*\ \text{P2616}\ ((\text{P1945}\mid\text{P2068})^+\ \text{P2616})^*)^{\leq 1}(\text{P1945}\mid\text{P2068})^*\mathcal{S}$
f_6	$\mathcal{C}\mathcal{P}^*$ P2068$^{\geq 2}$ $\mathcal{P}^*\mathcal{S}$	$\mathcal{C}((\text{P1945}\mid\text{P2616})^*\ \text{P2068}^{\leq 2}\ ((\text{P1945}\mid\text{P2616})^+\ \text{P2068}^{\leq 2})^*)^{\leq 1}(\text{P1945}\mid\text{P2616})^*\mathcal{S}$
f_7	C1945 P2068 $\mathcal{P}^*\mathcal{S}$	C2068 $\mathcal{P}^*\mathcal{S}$

Table 2: Set mapping of f_i

Notice also that each of these function is a valid reduction function, in that it satisfies the *invariance* and the *progress* properties. Invariance was established in a previous paper [7] using manual proofs; progress holds because, for every rewrite rule f_i, it is true that $f_i(\mathcal{A}) \subsetneq \mathcal{A}$.

Recall Equation 5, which is the glue of our methodology's strategy. It demands that for each f_i, we find:

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

We denote such a set induced by f_i as A_i. Using the above-described supports and the sets they are mapped to, Table 3 presents these in regular expression form for each of f_1 through f_7. In the first two, we present this set first in explicit terms, then in terms of its negation removed from $\mathcal{A}$; subsequent sets are presented only in terms of the removal of their negations, as this form is much more readable.

A_i	$(\mathcal{A} - \mathsf{support}(f_i)) \cup f_i(\mathsf{support}(f_i))$
A_1	$\mathcal{C}$(P2068|P2616)$^*\mathcal{S}$ *i.e.*, $\mathcal{A} - \mathcal{CP}^*$ P1945 $\mathcal{P}^*\mathcal{S}$
A_2	$\mathcal{C}$((P2068|P2616)*P1945)$^{\leq 1}$((P2068|P2616)$^+$P1945)*(P2068|P2616)$^*\mathcal{S}$ *i.e.*, $\mathcal{A} - \mathcal{CP}^*$ P1945 P1945 $\mathcal{P}^*\mathcal{S}$
A_3	$\mathcal{A} -$ C2616 P2616 $\mathcal{P}^*\mathcal{S}$
A_4	$\mathcal{A} - \mathcal{CP}^*$ P2616 S2616
A_5	$\mathcal{A} - \mathcal{CP}^*$ P2616 P2616 $\mathcal{P}^*\mathcal{S}$
A_6	$\mathcal{A} - \mathcal{CP}^*$ P2068 P2068 P2068 $\mathcal{P}^*\mathcal{S}$
A_7	$\mathcal{A} -$ C1945 P2068 $\mathcal{P}^*\mathcal{S}$

Table 3: Sufficient subspaces (A_i) under each of $f_1, \ldots, f_7$

Taking the intersection of these sufficient subspaces should give us the minimal sufficient subspace supported by the given reduction functions. While we could continue to represent our sets using regular expressions, unions across these sets will quickly become difficult to read in such a form; as such, we will henceforth represent sets using the corresponding finite-state automata. We also note that these automata correspond precisely with our notion of an ACN as presented in Section 2.1, where every complete path in that ACN represents a "member" CCN.

The ACN (automaton) for $\mathcal{A}$ is a simple nine-state machine with three start states each with no inbound edges (C1945, C2068, and C2616), three end states with no outbound edges (S1945, S2068, and S2616), and three intermediary states (P1945, P2068, and P2616) 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 is illustrated in Figure 1.

For clarity, we will omit the $\mathcal{CS}$ edges from diagrams below; none of these edges are removed by any of our reduction rules.

The reduction of this ACN by taking its intersections with the sets represented by A_1, A_2, A_3, A_4, A_5, and A_7 is a trivial operation, and gives rise to the ACN shown in Figure 2.

The intersection of this set with $\mathcal{A}_6$ is more complicated; it removes a subsequence of length 3, meaning simply removing an edge from our ACN will not suffice. Instead, it requires the introduction of a second P2068 state. This is

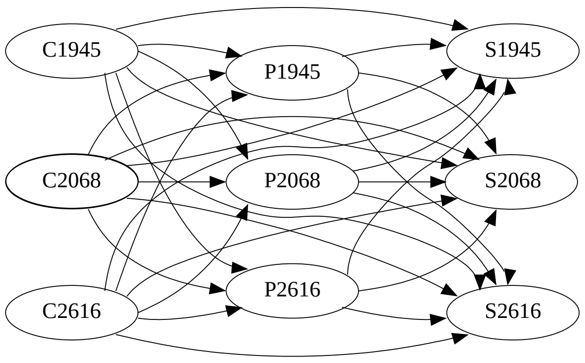

Figure 1: Automaton for $\mathcal{A}_0 = \mathcal{A}$

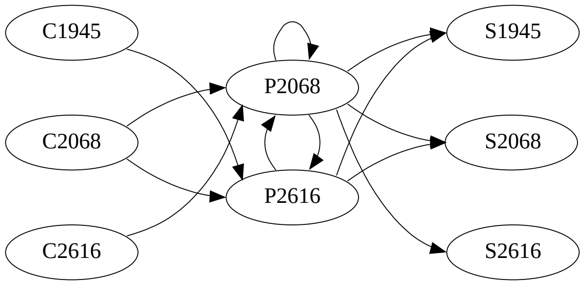

Figure 2: Automaton for $A_1 \cap A_2 \cap A_3 \cap A_4 \cap A_5 \cap A_7 - \mathcal{CS}$

done in Figure 3, which shows the automaton for $\mathcal{A}_7$, the minimal sufficient subset of $\mathcal{A}$ under our seven reductions $R_1, \ldots, R_7$.

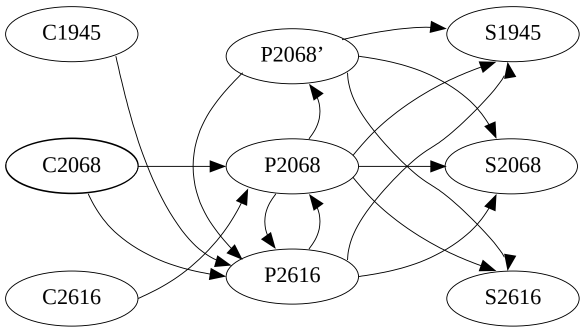

Figure 3: Automaton for $\mathcal{A}_7 - \mathcal{CS}$, *i.e.*, $A_1 \cap \cdots \cap A_7 - \mathcal{CS}$

Unfortunately, this is not an acyclic automaton; thus, the language it represents is still infinite. The two remaining cycle paths are P2616 $\rightarrow$ P2068 $\rightarrow$ P2616 and P2616 $\rightarrow$ P2068 $\rightarrow$ P2068$'$ $\rightarrow$ P2616; to remove them both, either the edge P2616 $\rightarrow$ P2068 must be removed, or both of P2068 $\rightarrow$ P2616 and $Pb' \rightarrow Pc$ must be removed, or some offset of both of the two loop patterns (or some multiples thereof) must be removed.

While we are not yet at our goal, we have now significantly narrowed the space of unexamined potential reductions which need exploring; since the original reduction set was devised as a set of *ad hoc* observations, it should not surprise us that other valid reductions might in fact exist. Using this small "problematic" search subspace, we focused our examination efforts upon the particular agent sequences described by it, and discovered an additional valid reduction which we present here as R_8.

Stated in the style of [6], R_8 is:

- Consider the arrangement $x \rightarrow$ P2068 $\rightarrow$ P2616 $\rightarrow$ P2068 $\rightarrow y$. The passive behavior of P2616 will never initiate a Continue message of its own, neither will it add any expectation to the upstream path which was absent at the downstream P2068; its behavior is end-to-end, and thus it will never block an inbound message; furthermore, since both P2068 and P2616 self-identify as HTTP/1.1, it will have no effect upon the perceived versions of messages received by either P2068. Therefore, this arrangement is semantically equivalent to one in which the middle P2616 is removed.

This gives rise to the rewrite rule:

$$R_8 : x \text{ P2068 P2616 P2068 } y \rhd_{R_8} \{x \text{ P2068 P2068 } y\} \tag{6}$$

Which, when converted to the reduction function $f_8 = f_{R_8}^{(*)}$, has for support the set $\mathcal{CP}^*$ P2068 P2616 P2068 $\mathcal{P}^*\mathcal{S}$ which it maps to $\mathcal{C}$ ((P1945|P2616)*|(P2068$^+$(P2616 (P1945|P2616))$^{\leq 1}$))*((P1945|P2616)* P2068 P2616$^{\leq 1}$)$^{\leq 1}\mathcal{S}$. This implies a sufficient subspace A_8 of:

$$A_8 = \mathcal{A} - \mathcal{CP}^* \text{ P2068 P2616 P2068 } \mathcal{P}^*\mathcal{S} \tag{7}$$

The intersection of this set with the remaining $A_1 \cap \cdots \cap A_7$ is presented in Figure 4.

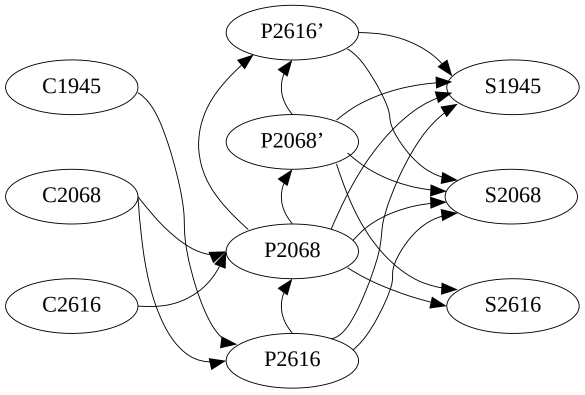

Figure 4: Automaton for $\mathcal{A}_8 - \mathcal{CS}$, *i.e.*, $A_1 \cap \cdots \cap A_8 - \mathcal{CS}$

Or, as a regular expression,

$$(\text{C1945 P2616} \mid \text{C2068 P2616}^{\leq 1} \mid \text{C2616})\ \text{P2068}\ \ \text{P2068}^{\leq 1}(\text{S2616} \mid \text{P2616}^{\leq 1}(\text{S1945} \mid \text{S2068})) \mid \mathcal{CS} \tag{8}$$

This is an acyclic automaton, and similarly, the regular expression has no unbounded repetition. As such, we have achieved our goal and found a finite $\mathcal{A}_n = A_1 \cap \cdots \cap A_8$ which is a sufficient subset of $\mathcal{A}$. The set $\mathcal{A}_8$ has 49 member strings[4]; thus, it is sufficient to compute $\pi(a)$ for only these 49 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 3.1. *Given the infinite language of arrangements of HTTP agents $\mathcal{A} = \mathcal{C}\mathcal{P}^*\mathcal{S}$ with $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{S}$ defined as in the opening paragraph of Section 3.1, there exists a finite subset $\mathcal{A}_\perp$ of $\mathcal{A}$ with 49 members, expressed by Equation 8, such that if all the members of $\mathcal{A}_\perp$ are deadlock-free, then all the members of $\mathcal{A}$ are deadlock-free; furthermore, whether any $a \in \mathcal{A}$ is deadlock-free can be computed without having to model it directly by finding the set $B \subseteq \mathcal{A}_\perp$ such that whether a is deadlock-free is precisely the logical* AND *of the set $\{\pi(b) \mid b \in B\}$.*

While it is not the case that all $a \in \mathcal{A}_\perp$ are deadlock-free, that particular observation motivates one generalized application of this methodology: given a verifiably correct system, new revisions of agents can be introduced as fully "backward compatible" when and if sufficiently many reductions have been proven and sufficiently many models have been verified so as to ensure that the new infinite set of possible configurations remains fully correct. This would constitute a sufficient condition to ensure backward-compatibility.

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 [6], we proposed a pair of "failure patterns" which induce all failure cases explored in that paper. Here we argue for the sufficiency of a generalized form of these failure patterns to detect all deadlock cases for all arrangements in $\mathcal{A}$.

In [6], we speculated that all $a \in \mathcal{A}$ such that $\pi(a) =$ **false** will match one of these two patterns:

$$((\mathcal{C}\ \mathcal{P}^*\ \text{P2068}) \mid \text{C2068})\ (\text{P2068} \mid \text{P2616})\ (\text{S1945} \mid \text{P1945}\ \mathcal{P}^*\ \mathcal{S}) \tag{9}$$

$$(\text{C2068} \mid \text{C2616})\ (\text{P2068} \mid \text{P2616})^*\ \text{P2068}\ (\text{S1945} \mid \text{P1945}\ \mathcal{P}^*\ \mathcal{S}) \tag{10}$$

We have already shown in [6] that no arrangement matching these patterns can be reduced to any deadlock-safe arrangement, and have shown that these patterns do not match any member of $\mathcal{A}_\perp$ which are deadlock-safe. The correctness of the rule can therefore be established by showing that no arrangement which does not match any of these patterns can be reduced to an arrangement which does, *i.e.*, by showing the *closure* of these patterns as a segregator of $\mathcal{A}$ under the reduction operators. We disprove the proposed pattern's exhaustive correctness by showing three counterexamples, in which classes of arrangements which do not match either failure pattern can be reduced to arrangements which do.

- Consider equivalence R_5, which can be written as x P2616 $y \leftrightarrow x$ P2616$^+$ y and the class of arrangements $(\mathcal{C}\ \mathcal{P}^*\ \text{P2068} \mid \text{C2068})\ \text{P2616}\ (\text{S1945} \mid \text{P1945}\ \mathcal{P}^*\ \mathcal{S})$ (a subset of pattern 9; intuitively, arrangements in which either a C2068 or a P2068 speaks with a S1945 or equivalent proxy via a single P2616). This production rule allows us to change that single P2616 into an arbitrarily long sequence of P2616s; in doing so, we clearly create

[4]A result of 53 was cited without proof in [6] in referring to this work, which reflected an earlier version of this result.

an arrangement which matches neither failure pattern (pattern 1 only allows for a single P2616 immediately preceding the S1945, and pattern 2 demands a P2068 in the final position before the S1945).

- Consider equivalence R_7, which can be written as C2068 $x \leftrightarrow$ C1945 P2068 x and the class of arrangements C2068 $($P2068 $|$ P2616$)^*$ P2068 $($S1945 $|$ P1945 $\mathcal{P}^*$ $\mathcal{S})$ (a subset of pattern 10; intuitively, arrangements in which a C2068 speaks via a series of HTTP/1.1 proxies with a P2068 immediately downstream of a S1945 or equivalent proxy). This production rule allows us to transform the leading C2068 into any member of the set $\mathcal{C}$ $\mathcal{P}^*$ P2068; clearly this includes cases in which the client is not C2068, and therefore pattern 10 fails to match; furthermore, arrangements like C2068 P2616 P2068 S1945 (which matches pattern 10) is equivalent (by R_7) to arrangements like C1945 P2068 P2616 P2068 S1945 which matches neither of the failure patterns.

- Consider equivalence R_8, which can be written as

  $$x \text{ P2068}^2\ y = x \text{ P2068 P2616 P2068 } y$$

  and the class of arrangements $\mathcal{C}$ $\mathcal{P}^*$ P2068 P2068 $($S1945 $|$ P1945 $\mathcal{P}^*$ $\mathcal{S})$ (a subset of pattern 9; intuitively, an arrangement in which a sequence of two P2068 agents is immediately downstream of the S1945 or equivalent proxy). As with R_7, this production can cause arrangement substrings like P2068 P2068 S1945 (which could match either pattern) to be transformed into substrings like P2068 P2616 P2068 S1945 which can only potentially match pattern 10, and that only if the arrangement starts with C2068 (which mayt not be the case, since the set we are considering may begin with any member of $\mathcal{C}$).

In light of these three failure conditions, we have re-formulated the two failure patterns so as to be closed with respect to $R_2, \ldots, R_8$. The new patterns are stated as Theorem 3.2.

Theorem 3.2. *All $a \in \mathcal{A}$ such that $\pi(a) =$ **false** will match at least one of these regular expressions:*

$$((\mathcal{C}\ \mathcal{P}^*\ \text{P2068}) \mid \text{C2068})\ (\text{P2068} \mid \text{P2616}^+)\ (\text{S1945} \mid \text{P1945}\ \mathcal{P}^*\ \mathcal{S}) \tag{11}$$

$$((\mathcal{C}\ \mathcal{P}^*\ \text{P2068}) \mid \text{C2068} \mid \text{C2616})\ (\text{P2068} \mid \text{P2616})^*\ \text{P2068}\ (\text{S1945} \mid \text{P1945}\ \mathcal{P}^*\ \mathcal{S}) \tag{12}$$

Proof. Among $a \in \mathcal{A}_\perp$, 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 $\mathcal{A}_\perp$ which are deadlock-prone, and similarly, all members of $\mathcal{A}$ which are deadlock-safe are reducible to members of $\mathcal{A}_\perp$ which are deadlock-safe. As such, the correctness of this theorem rests upon three properties: first, that for any member $a \in \mathcal{A}_\perp$, $\pi(a) =$ **false** iff a is in the union of these patterns (that is, the patterns correctly identify $\mathcal{A}_{\textbf{false}} \cap \mathcal{A}_\perp$; second, that any member of the union of these two patterns is reducible to a member of $\mathcal{A}_{\textbf{false}} \cap \mathcal{A}_\perp$; third, that no arrangement $a \in \mathcal{A}$ which does not match either of these patterns can be reduced to one which does. These three properties are proven in Appendix A as Lemmas A.1, A.2, and A.3, respectively. □

3.2 HTTP Connection Management

One early HTTP protocol optimization was the introduction of persistent connections [24]. This feature allows multiple transactions to be sequentially conducted over a single transport (TCP) connection, amortizing the cost of TCP connection setup and teardown. This feature is the cornerstone of the *connection management* features introduced in HTTP/1.1 [21], which standardizes the negotiation and use of this feature. However, non-standardized variants of the feature were also introduced by industry and were implemented in their HTTP/1.0 agents; most significantly, Netscape included both a HTTP/1.1 conforming `Connection` header and a privately defined[5] `Proxy-Connection` header starting with its Navigator 1.1 product, which was an HTTP/1.0 implementation. As a result, both RFCs specifying the HTTP/1.1 specification include provisions to allow interoperation with these features in HTTP/1.0 agents under a particular set of "safe" circumstances [11, §19.7.1][12, §19.6.2].

Since persistent connections are a hop-by-hop feature (*i.e.*, something which should be carried out entirely between two immediately adjoining nodes and having no effect on any agents upstream or downstream of the pair), one would hope that the protocol could be modeled simply in terms of a two-process client-server interaction. Unfortunately, HTTP/1.0 lacked a mechanism for discriminating between hop-by-hop and end-to-end headers; as such, it is possible for a "pure" (no persistent connection support) HTTP/1.0 proxy to send requests to its immediately-upstream agent which a "pure" (non-PC) HTTP/1.0 client would never send (namely, requests which transparently pass along the persistent connection request token from an agent downstream of that proxy). As such, we must concern ourselves with compositions of HTTP agents with respect to their persistent connection behavior.

We have constructed simple PROMELA models for use in the SPIN verifier which capture a highly-distilled version of persistent connections as defined by RFCs 1945 and 2616, as well as a version of the `Proxy-Connection` header employed in HTTP/1.0 versions of Netscape Navigator. We imagine a set of 15 interesting agent models: 5 clients, 6 proxies, and 4 servers as listed in Figure 5. Each individual model is less than 100 lines of PROMELA code.

While we may wish to prove properties such as "civility" of shutdowns (anyone who knows how to interpret a "close" signal does so gracefully), for this paper we concern ourselves with the one potential deadlock case: the case in which a client expects a connection to close (marking the end of a response) which its immediate upstream agent will not send because it is waiting for another request. Naturally, in the wild this is not a true deadlock *per se* because all agents acting as servers (whether origin or proxy) will attach timeouts to connections with their clients; still, tying up connection resources sleeping unnecessarily is an undesirable property with negative performance effects [2, 1], so we omit that timeout from our models and treat that situation as a correctness violation, detected by our model as a deadlock.

[5]To the best of our knowledge, the only publicly available documentation of Netscape's `Proxy-Connection` header is the source code of early versions of the Mozilla open-source browser.

- *client-10*: pure HTTP/1.0 client
- *client-10-c*: speaks 1.0 keepalive
- *client-10-cpc*: also speaks 1.0 proxy-keepalive hack
- *client-11*: pure HTTP/1.1 client
- *client-11-c*: speaks 1.0 keepalive with 1.0 servers only
- *proxy-10*: pure HTTP/1.0 proxy
- *proxy-10-c*: understands and speaks 1.0 keepalive, oblivious to proxy-keepalive hack (passes through)
- *proxy-10-cpc*: understands and speaks 1.0 keepalive and proxy-keepalive hack
- *proxy-11*: pure HTTP/1.1 proxy - disregards all 1.0 keepalives; oblivious to proxy-keepalive hack (passes through)
- *proxy-11-c*: understands and speaks 1.0 keepalive downstream; upstream with 1.0 servers only; oblivious to proxy-keepalive hack (passes through)
- *proxy-11-cpc*: understands and speaks 1.0 keepalive downstream; upstream with 1.0 servers only; filters out proxy-keepalive
- *server-10*: pure HTTP/1.0 server
- *server-10-c*: understands 1.0 keepalive; ignores `Proxy-Connection`
- *server-11*: pure HTTP/1.1 server
- *server-11-c*: understands 1.0 keepalive; ignores `Proxy-Connection`

Figure 5: PROMELA models of HTTP Persistent Connections

Given this set of models, we have a new $\mathcal{A} = \mathcal{C}\mathcal{P}^*\mathcal{S}$ where

$$\begin{aligned}
\mathcal{C} &= \{\textit{client-10, client-10-c, client-10-cpc, client-11, client-11-c}\}, \\
\mathcal{P} &= \{\textit{proxy-10, proxy-10-c, proxy-10-cpc, proxy-11, proxy-11-c, proxy-11-cpc}\}, \text{ and} \\
\mathcal{S} &= \{\textit{server-10, server-10-c, server-11, server-11-c}\}.
\end{aligned}$$

We also have a new π function which can be computed using the SPIN model checker for any $a \in \mathcal{A}$, where $\pi(a) =$ **true** means that arrangement a is persistent-safe (the above-described deadlock/timeout will not arise) and $\pi(a) =$ **false** means that arrangement a is persistent-unsafe (the above-described deadlock/timeout may occur under certain conditions).

From the assumption that HTTP/1.1 persistent connections are safe between any two immediately-adjoining HTTP/1.1 agents (an assumption which can be verified using simpler models using only a client-server arrangement) and that all client-server pairings are safe (also easily verified) and careful analysis of our models, we have derived a (non-exhaustive) set of 29 rewrite rules which preserve this π while rewriting $\mathcal{P}^+$ sub-terms. These rules are listed in Appendix B.1.

Under these reductions, we produce an ACN which includes 70 possible $\mathcal{P}^*$ values, each of which must be tested against the 20 $\mathcal{C} \times \mathcal{S}$ pairings. While 1400 is a large number of models to check, it is far from intractable (particularly if we allow ourselves to use probabilistic methods like supertrace/bitstate hashing). One should also note that the list of reductions presented here is far from exhaustive (this set was derived in the course of an afternoon using *ad hoc* methods).

3.3 An Active Networking MPEG Router

In this section, we show that our framework can be used not only to check the correctness of protocols, but as a way of defining design and verification constraints which will ensure correct interoperation with arbitrary network environments.

Some applications of active networking lend themselves very naturally to finite-state model verification because they are designed to be both algorithmically simple and to rely upon a minimal amount of application state. By their nature we would also expect for them to be deployed into a network in which they will be interacting with other active network applications as well as conventional routers and hosts (*i.e.*, we expect that they will be composed with other processes); it seems reasonable to believe then that such applications are good candidates for this methodology.

Consider the method for handling MPEG flows proposed in [14], which attempts to do intelligent dropping of MPEG frames based upon the dependency and priority relationships between the three classes of MPEG frames (I, P, and B frames). 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.

These relationships suggest simple packet-dropping rules: If at all possible, dropping I frames should be avoided; Once a B frame has been dropped, all successor B frames until the next P frame are useless and should be dropped; Similarly, once a P frame has been dropped, all successor packets can be safely dropped until the next I frame. The applet presented in [14] implements a simple version of this algorithm requiring constant time and storage.

Unfortunately, the algorithm as implemented relies upon the router seeing the packets constituting an MPEG stream in original frame order. This is an optimistic assumption given that packet reordering is a common phenomenon in the Internet [25]; indeed, it is not hard to devise pathological reorderings among pairs of sequentially adjoining packets which can cause the applet to erroneously treat large numbers of frames as undecodable and therefore discardable. The algorithm also implicitly assumes that the stream reaches it intact; 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 the terms of our methodology. 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; it therefore makes sense for us to ask within our framework whether the composition of a packet-reordering network with such a router (*server*, *reord-net*, *mpeg-router*, *client*) 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 (*server*, *mpeg-router*, *mpeg-router*, *client*) 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 (*server*, *reord-net*, *mpeg-router*, *reord-net*, *mpeg-net*, *client*)? What of composition 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 correctness property π which identifies (using any verification technique) whether packets can be erroneously dropped by any particular combination of those components.[6]

3.3.1 Reducibility as Specification

Rather than thinking in terms of reductions/rewrite rules which a particular fully-specified application induces, one may prefer to state a design goal of an application in terms of a set of reductions. For example, we could designate a set of reductions which we must be able to prove our application agrees with. Some reductions will naturally arise as properties of the existing network infrastructure; For example:

- R_1 : x *reord-net reord-net* $y \ \triangleright\ x$ *reord-net* y
- R_2 : x *drop-net drop-net* $y \ \triangleright\ x$ *drop-net* y
- R_3 : x *drop-net reord-net* $y \ \triangleright\ x$ *reord-net drop-net* y

Notice that the third rule forces a particular associativity upon adjoining pairs of *reord-net* and *drop-net* nodes; in so doing, these three rules together form a compound rule which says any sequence which exclusively contains both *drop-net* and *record-net* can be modeled by "*reord-net drop-net*".

Ultimately, we would like to be able to say something about the *mpeg-router* in any network situation. Assuming that all the relevant characteristics of intervening networks can be represented using models of *record-net* and *drop-net*, we can then define $\mathcal{A}$ much as before as $\mathcal{S}\mathcal{P}^*$ *mpeg-router* $\mathcal{P}^*\mathcal{C}$, where

$$\begin{aligned} \mathcal{S} &= \textit{server} \\ \mathcal{P} &= \textit{mpeg-router},\ \textit{reord-net},\ \textit{drop-net} \\ \mathcal{C} &= \textit{client} \end{aligned}$$

This definition has the advantage that it excludes all network configurations which do not include an *mpeg-router*, that is, all arrangements in which we are not interested. Given the already-stated three reductions, we can derive $\mathcal{A}_3$ (pictured in Figure 6) using their corresponding f_is. $\mathcal{A}_3$ is still an infinite language; *mpeg-router* can cycle back to itself, or cycle between itself and either of *record-net* or *drop-net*, or can cycle through the *reord-net drop-net* sequence back to itself.

We can then state, as an engineering goal, that *mpeg-router* should provably satisfy a set of reductions which yield a finite $\mathcal{A}_n$. The choice of proof method is not particularly important to us for the moment; whichever is best suited to the design and development environment can equally well be used. Perhaps it could be done using a finite-state-machine equivalence/inequivalence test (an NP-complete problem), or using creative applications of type theory [8]. Regardless, the following reductions would be sufficient, and make for an illustrative example of target reductions which could be set as evaluation criteria for an *mpeg-router*:

[6] Notice that the MPEG router is itself allowed to drop packets when its link load is too high. For modeling purposes this means that whenever the router wants to forward a packet there are two possible outcomes: A correct forwarding or a "legal" drop. It is not really useful for us to consider a notion of "how many" drops are acceptable - a drop is either correct (acceptable) because of link overload (which may arise at arbitrary times) or previous legal drops, or it is illegal (the packet was still valuable and the link was not overloaded). A different formulation of π could include a notion of drop rate, or anything else of interest; bounded delay and jitter also come to mind as useful properties.

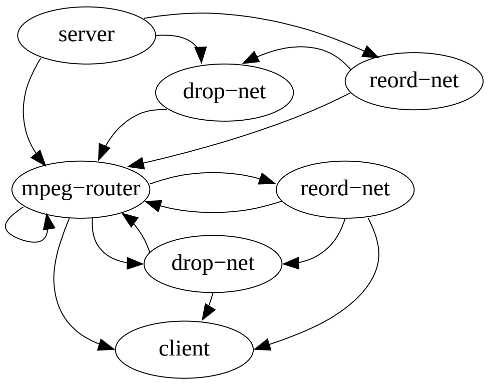

Figure 6: ACN of $\mathcal{A}_3$ for an MPEG-routing network

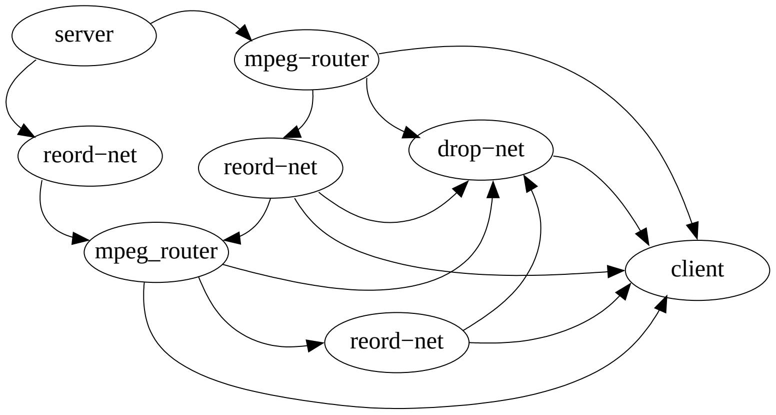

Figure 7: Acyclic ACN of $\mathcal{A}_6$ for an ideal MPEG router, excluding all paths which contain no *mpeg-router*

- R_4 : x *mpeg-router mpeg-router* y $\rhd$ $\{x$ *mpeg-router* $y\}$
- R_5 : x *drop-net mpeg-router* y $\rhd$ $\{x$ *mpeg-router drop-net* $y\}$
- R_6 : x *reord-net mpeg-router reord-net mpeg-router* y $\rhd$ $\{x$ *reord-net drop-net mpeg-router* $y\}$

If all six reductions are valid, then $\mathcal{A}_6$ has 16 members, and only 12 of these include an *mpeg-router* component; thus, by testing only those 12, we would establish the behavior of *mpeg-router* in all possible network configurations. These 12 are expressed in the ACN in Figure 7.

3.3.2 Algorithms Resilient to Network Anomalies

We now discuss modified versions of the algorithm from [14] which are more resilient to common network anomalies. For all of our modifications we require a change to the MPEG packet header format used in that work[7]: we add one integer field, "`dep_frame_no`". In an I-frame, this field has the same value as `frame_no`. For 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

[7]Notice that there is no current standard format for streaming MPEG packets over the Internet; as such, any algorithm at this point will be format-dependent.

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) interchangably.

Missing Packets If the packet stream remains in order but may lose packets before reaching the *mpeg-router*, 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:

1. If it is an I-frame, that frame is forwarded as usual; this always sets the system to a "normal" state.
2. If it is a P-frame:
   (a) If the `frame_no` of the last forwarded packet is less than `dep_frame_no`, drop all packets until the next I-frame arrives; return to "normal" state.
   (b) Otherwise, forward the frame as usual, return to "normal" state.
3. If it is a B-frame:
   (a) If the `frame_no` of the last forwarded packet is less than `dep_frame_no`, drop all packets until the next I-frame arrives; return to "normal" state.
   (b) Otherwise, drop all B-frames until either an I-frame arrives (treat as above) or the next P-frame arrives; process that P-frame according to the above rule.

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 preemptive drop strategy accordingly. This processing would precede all local policy decisions, including congestion-driven drops; acting as such, it prevents any "worthless" packet from even reaching the routing logic.

Using this previous-drop-optimizing version of the *mpeg-router*, we can easily prove the stated target reductions R_4 and R_6. However, R_5 does not hold because the right-side ordering (with *drop-net* last) would allow a "wasteful" MPEG stream to be fed to y, while the left-side ordering (with *mpeg-router* last) would always feed an "efficient" MPEG stream (every packet is "useful") to y. This leaves us with the ACN shown in Figure 8, which is still cyclic.

Reordered Packets For each move, 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 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 our 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

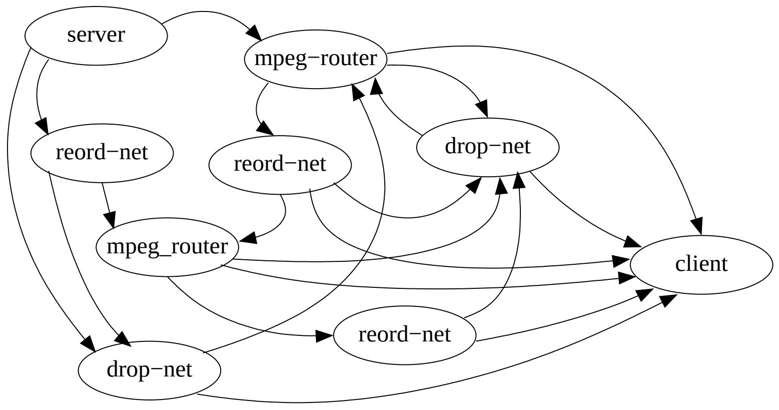

Figure 8: ACN for $\mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 \cup \mathcal{A}_4 \cup \mathcal{A}_6$ (missing packets *mpeg-router*)

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.

As each packet arrives, it is processed according to the following algorithm:

- I-frame:
  1. If too congested, drop and enter in `drop_i_log`.
  2. otherwise, forward as usual.
- P-frame:
  1. If `dep_frame_no` $\in$ `drop_i_log` $\cup$ `drop_p_log`, drop and enter into `drop_p_log`.
  2. otherwise, if too congested, drop and enter in `drop_p_log`.
  3. otherwise, forward as usual.
- B-frame:
  1. If `dep_frame_no` $\in$ `drop_p_log`, drop and enter into `drop_b_log`.
  2. otherwise, if any frame number in `drop_b_log` is between `dep_frame_no` and `frame_no`, drop and enter into `drop_b_log`.
  3. otherwise, if too congested, drop and enter in `drop_b_log`.
  4. otherwise, forward as usual.

This algorithm guarantees that the mpeg-router 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 naive 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 reorderings (as opposed to the incorrect drops and incorrect retentions which the original could induce).

Using this reordering-resilient version of *mpeg-router*, we can easily prove the stated target reductions R_4, R_5, and R_6. This gives rise to the acyclic ACN shown in Figure 7; as stated above, this ACN represents a set of 12 arrangements which fully characterize $\mathcal{SP}^*$ *mpeg-router* $\mathcal{P}^*\mathcal{S}$.

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 in all likelihood more costly than simply forwarding packets which would eventually be worthless. Such an approach is then clearly not amicable to a stateless/low-state routing mechanism; for this reason, we feel our proposed reordering-resilient algorithm is much better suited to this application.

3.4 Web Intra-Cache Consistency

There is nothing in this 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, 5], a protocol whose properties can be easily established by way of direct proof. We will also show how a realistic model of web caching gives rise to situations which are beyond the expressive power of our current framework.

For BTC, the interesting π property to define is whether the client at the end of some arrangement in the ACN $\mathcal{SP}^*\mathcal{C}$ will be guaranteed to see an (internally) consistent sequence of responses, *i.e.*, one which is temporally non-decreasing.[8] If $\pi(a) =$ **true**, then arrangement a will always cause the client's view of the server to be consistent (temporally/causally non-decreasing); $\pi(a) =$ **false** indicates that arrangement a may 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 (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 meaningful arrangements.

[8]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.

A simplified model of the web[9] can be represented with respect to BTC using the following agents:

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

where *proxycache-btcpush* uses the end-to-end strong consistency extension defined in [4]. 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 closes 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.

First, we note that 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* agents having no effect or indifference to the ordering of *proxy-scrubber* and *proxycache-plain* agents. These are reflected as reductions R_1 through R_6 in Appendix C.1. The definition and correctness of BTC itself gives rise directly to 14 additional reductions, R_7 through R_{20} also in Appendix C.1.

From these twenty reductions we, through the application of our methodology, identify a characterization-sufficient set/ACN $\mathcal{A}_\perp$ containing four (4) member arrangements, described by this regular expression:

$$\mathcal{A}_\perp = (\textit{server-plain}|\textit{server-btc})\ \textit{proxycache-plain}^{\leq 1}\ \textit{client} \tag{13}$$

Intuitively, this result tells us that any arrangement of proxying agents will ultimately map down (from a consistency-correctness point of view) to four basic configurations: the two basic kinds of servers, each in the presence of either no caches or a single plain (non-BTC) proxy cache. The value of π for these four configurations is trivial; clearly, the case without a *proxycache-plain* will always be consistent, and the case with it will always be inconsistency-prone. Thus, our reduction rules provide us with a simple strategy to assess the consistency-safety of any proxy-cache arrangement.

A π Decision Rule Using Correctness Patterns As in Theorem 3.2, 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. Formally, we state these three rules in Theorem 3.3 as Equations 14, 15, and 16.

[9]Our formalism is designed to examine linear compositions of agents. However, the routing of requests in a web caching network may not be such a linear composition; a given agent may have a choice of multiple upstream agents which are all able to answer its request, leading to a divergence in the network. This is discussed in greater depth in Section 4.1.

Theorem 3.3. *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 at least one of these three patterns:*

$$(\text{server-plain} \mid \text{server-btc})\ (\text{proxy-scrubber} \mid \text{proxy-plain})^{*}\ \text{client} \tag{14}$$

$$\text{server-btc}\ \mathcal{P}^{*}_{\textit{clean}}\ \text{proxycache-btc}\ (\text{proxy-plain} \mid \text{proxy-scrubber})^{*}\ \text{client} \tag{15}$$

$$\text{server-btc}\ \mathcal{P}^{*}_{\textit{clean}}\ \text{proxycache-btcpush}\ \mathcal{P}^{*}\ \text{client} \tag{16}$$

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

Proof. Similar to Theorem 3.2. A “safety pattern” is simply the compliment of a “failure pattern”, so its validity is established by the same properties: first, it will correctly partition $\mathcal{A}_{\perp}$; second, it will define a set which is closed under all reductions (that is, reductions preserve both membership and non-membership).

These properties are established in Appendix C.2 as Lemmas C.1 and C.2. $\square$

4 Conclusions

In this paper, we have presented a method by which an infinite set of arrangements (compositions of processes) can be modeled and verified using a finite set of compositions. This methodology relies upon the discovery of a sufficient set of reduction relationships which establish causal equivalence between particular subsequences of processes. We have applied this methodology to our previous research into a particular deadlock property of the HTTP protocol to show that all possible arrangements of HTTP agents which could give rise to this deadlock can be characterized in terms of a set of 49 particular finite-length arrangements. We have also presented a similar result with respect to the safety of HTTP’s persistent connection feature. We then proposed an application of our methodology as a way of defining correctness for a proposed active-networking protocol component, and discussed how our methodology needs to be extended to apply to more general compositional topologies such as trees and DAGs.

4.1 Future Work

Algorithmic Complexity It is interesting to consider what the complexity bounds of the various operations discussed in this paper may be; in particular, the complexity of the application of a rewrite rule, the bound on how many rewrites must be performed to reduce a string $a \in \mathcal{A}$ to a subset of $\mathcal{A}_{\perp}$, the complexity of translating a rewrite rule into the necessary $\mathsf{support}(f)$ and $f(\mathsf{support}(f))$ sets, and the complexity of finding A_n given a list of n such sets.

Caching Networks with More General Topologies Since BTC is an intra-cache property, its correctness is with respect to the output of a single cache. The content (and therefore output) of any given cache is in turn a function of the upstream web network via which it retrieves content from the origin server.

Thus far we have considered only the case of a linear delivery network. For this application's purpose, this is sufficient to model the trivial case of a linear sequence of proxies. It is also sufficient to model the case of a network of proxies forming a logical tree rooted at the origin server, because there is only one path in the graph from the origin server to any given cache; as such, a local property of any given cache is purely a function of that linear upstream network.

In the general case, it may be possible for other graph structures to occur. A particular agent may have several upstream proxies 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 some cache. Such a network can be represented as a DAG[10] with the interesting subgraph consisting of the set of vertices and edges which reach the particular cache in question (and ergo any client speaking directly with that cache).

Our methodology as we have presented it thus far has not been formulated to deal with such structures. However, BTC definitionally gives rise to certain reductions which could apply to such structures; for example, if there are no *proxy-scrubber*s upstream (along any of the diverging paths) of a *proxycache-btc* agent, then that agent's connection to its upstream DAG can be correctness-modeled by replacing all inbound edges with an inbound edge from the origin (the $\mathcal{S}$ node). Similarly, if the root of the graph is a *server-btc*, then we can define the "hard reach" of BTC as all *proxycache-btc* nodes for which no *proxy-scrubber* exists on any path between the server and those nodes; we could then replace all of those nodes with a single "amalgamated" *proxycache-btc* node, connected to by the server node and connecting to every node which the original *proxycache-btc* nodes connected to.

Our formalism as it currently stands is simply not capable of handling this scenario. We have used tools analogous to a lexical analyzer, particularly context-free string matching and rewriting, based upon the assumption that the interesting structures are all linear. When manipulating a graph structure, however, a much stronger and more explicit notion of context is required, perhaps more analogous to that employed in grammatic parsing.

Broader Agenda We believe that this methodology 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 this methodology is a single step toward our goal of providing a 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.

Acknowledgements

S.D.G.

[10]A cycle could exist in a very poorly-behaved HTTP proxying network in which no member of the cycle implements the `Via` header. We consider such a case fundamentally pathological in that a client may simply never receive a request, and as such the question of correctness of cache consistency is moot; we therefore exclude the possibility from our model, as it is completely orthogonal to our concern.

References

[1] G. Banga and J. C. Mogul. Scalabel kernel performance for internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, 1998.

[2] Paul Barford and Mark Crovella. A performance evaluation of hyper text transfer protocols. Technical Report BUCS-TR-1998-016, Boston University Computer Science, 1998.

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

[4] 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.

[5] Adam D. Bradley and Azer Bestavros. Basis token consistency: Supporting strong web cache consistency. In *Global Internet Worshop*, Taipei, November 2002. (to appear).

[6] 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.

[7] 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.

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

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, pages 244–263, April 1986.

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

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

[12] 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.

[13] Patrice Godefroid. Partial-order methods for the verification of concurrent systems – an approach to the state-explosion problem. In *LNCS 1032*, January 1996.

[14] 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.

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

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

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

[18] Gerard J. Holzmann, Patrice Godefroid, and Didier Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proceedings of PSTV*, 1992.

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

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

[21] 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.

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

[23] Nancy Lynch and Frits Vaandrager. Forward and backward simulation – part II: Timing-based systems. *Information and Computation*, 121(2):214–233, September 1995.

[24] Jeffrey C. Mogul. The case for persistent-connection HTTP. In *Proceedings of ACM SIGCOMM '95*, August 1995.

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

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 (that is, the subset of $\mathcal{A}$ which match either or both of Equation 11 and Equation 12) 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 Equation 8:

$$(\text{C1945 P2616} \mid \text{C2068 P2616}^{\leq 1} \mid \text{C2616})\ \text{P2068}\ \ \text{P2068}^{\leq 1}(\text{S2616} \mid \text{P2616}^{\leq 1}(\text{S1945} \mid \text{S2068})) \mid \mathcal{CS}$$

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[11], 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 $\mathcal{A}_\perp \cap \mathcal{A}_{\textbf{false}}$. $\square$

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. The first four are derived from Equation 11, the last three from Equation 12.

1. $\mathcal{C}\ \mathcal{P}^*$ P2068 P2068 (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$)
2. $\mathcal{C}\ \mathcal{P}^*$ P2068 P2616$^+$ (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$)
3. C2068 P2068 (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$)
4. C2068 P2616$^+$ (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$)
5. $\mathcal{C}\ \mathcal{P}^*$ P2068 (P2068 | P2616)* P2068 (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$)
6. C2068 (P2068 | P2616)* P2068 (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$)
7. C2616 (P2068 | P2616)* P2068 (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$)

We now examine each of the 8 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 P1945 $y = x$ S1945, C1945 y

Consider an arrangement matching any of the seven patterns which contains a P1945. If that P1945 happens to correspond with the head of the (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$) sub-pattern (common to all patterns), then clearly at least the x S1945 produced arrangement will match the same original pattern; thus, R_1 preserves membership in the pattern space.

R_2 : x P1945$^+$ $y = x$ P1945 y

Consider an arrangement matching any of the seven patterns which contains one or more sequence of P1945s. Those P1945s are in one of two locations: either the leading $\mathcal{C}\ \mathcal{P}^*$ sub-pattern or the trailing (S1945 | P1945 $\mathcal{P}^*\ \mathcal{S}$) sub-pattern. In both cases, clearly the removal of some P1945s will not effect the match, since all P1945s 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.

[11]This was previously computed for arrangements of lengths 2, 3, and 4 in [6]. For arrangements of lengths 5 or 6 (*i.e.*, those containing 3 or 4 proxies), we have computed π using the supertrace/bitstate probabilistic optimization [16], which yields correct results with a very high probability and saves several orders of magnitude of compute time.

R_3 : C2616 P2616* $x =$ C2616 x

This reduction can only apply to arrangements matching patterns 1, 2, 5, and 7. In the case of 1, 2, and 5, the removed P2616s are matched by the $\mathcal{P}^*$ term, so clearly their removal will not effect the match. In the case of 7, the P2616s are matched by the $($P2068 | P2616$)^*$ sub-pattern, so clearly their removal will not effect the match. Thus, R_3 preserves membership in the pattern space.

R_4 : x P2616* S2616 $= x$ S2616

Consider an arrangement matching any of the seven patterns which contains the subsequence P2616* S2616. Clearly, this subsequence can only match the (common) $\mathcal{P}^*$ $\mathcal{S}$ sub-pattern, and thus the removal of the P2616s will not effect the match. Thus, R_4 preserves membership in the pattern space.

R_5 : x P2616$^+$ $y = x$ P2616 y

Consider each of the seven patterns:

1,3 P2616s can only appear in the $\mathcal{P}^*$ terms, so their removal will not effect the match.

2,4 P2616s appear in the $\mathcal{P}^*$ and P2616$^+$ terms, so the removal of those beyond the first will not effect the match.

5,6,7 P2616s appear in the $\mathcal{P}^*$ and $($P2068 | P2616$)^*$ sub-patterns, so their removal will not effect the match.

Thus, R_5 preserves membership in the pattern space.

R_6 : x P2068$^{\geq 2}$ $y = x$ P2068 P2068 y

Consider the seven patterns:

1,2 All P2068s 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 P2068 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 P2068 sequences of length greater than one must match within either the $\mathcal{P}^*$ P2068 $($P2068 | $Pc)^*$ P2068 sub-pattern or the $\mathcal{P}^*$ $\mathcal{S}$ sub-pattern; in the former case, removing all beyond the second P2068 will not effect the match (the sub-pattern demands at most the outer two P2068s), and in the latter case, removal of P2068s will not effect the match.

6,7 All P2068 sequences must match either the $($P2068 | P2616$)^*$ P2068 sub-pattern or the $\mathcal{P}^*$ term; in each case, the removal of P2068s beyond the second will not effect the match.

Thus, R_6 preserves membership in the pattern space.

R_7 : C1945 P2068 $x =$ C2068 x

Consider the seven patterns:

1 If the leading C1945 P2068 matches the $\mathcal{C}$ $\mathcal{P}^*$ sub-pattern, then the reduction has no effect. If the leading C1945 P2068 matches the $\mathcal{C}$ $\mathcal{P}^*$ P2068 sub-pattern, then the whole arrangement is rewritten to one of the form C2068 P2068 $($S1945 | P1945 $\mathcal{P}^*$ $\mathcal{S})$, 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

$$\text{C2068 P2616}^{+}\ (\text{S1945} \mid \text{P1945}\ \mathcal{P}^{*}\ \mathcal{S})$$

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

$$\text{C2068}\ (\text{P2068} \mid \text{P2616})^{*}\text{P2068}\ (\text{S1945} \mid \text{P1945}\ \mathcal{P}^{*}\ \mathcal{S})$$

which is precisely the set recognized by pattern 6.

Thus, R_7 preserves membership in the pattern space.

R_8 : x P2068 P2616 P2068 $y = x$ P2068 P2068 y

Consider the seven patterns:

1,2 The subsequence P2068 P2616 P2068 can only match either the $\mathcal{P}^{*}$ P2068 sub-pattern or the $\mathcal{P}^{*}$ term; in either case, removing the P2616 will not effect the match.

3,4 The subsequence P2068 P2616 P2068 can only match the $\mathcal{P}^{*}$ term, so removing the P2616 will not effect the match.

5 The subsequence P2068 P2616 P2068 can match either the $\mathcal{P}^{*}$ P2068, P2068 $(\text{P2068} \mid \text{P2616})^{*}$, or P2068 $(\text{P2068} \mid \text{P2616})^{*}$ P2068 sub-patterns, or one of the $\mathcal{P}^{*}$ terms, so in all cases the removal of the P2616 will not effect the match.

6,7 The subsequence P2068 P2616 P2068 can match the $(\text{P2068} \mid \text{P2616})^{*}$ or $(\text{P2068} \mid \text{P2616})^{*}$ P2068 sub-patterns, or the $\mathcal{P}^{*}$ term, so in all cases the removal of the P2616 will not effect the match.

Thus, R_8 preserves membership in the pattern space.

All reductions ($R_1 \ldots R_8$) preserve membership in the pattern space. $\square$

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_8. Equivalence R_1 tells us that we can treat the two sub-terms (S1945 and P1945 $\mathcal{P}^{*}$ $\mathcal{S}$) of the common closing sub-pattern $(\text{S1945} \mid \text{P1945}\ \mathcal{P}^{*}\ \mathcal{S})$ as being equivalent for the purposes of matching; thus, we do not treat the P1945 $\mathcal{P}^*$ $\mathcal{S}$ 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 S1945).

We now examine the inversions of f_2 through f_8 (corresponding with R_2 through R_8, 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 P1945 $y = x$ P1945$^+$ y

In all cases, additional P1945 are captured in a $\mathcal{P}^*$ term.

f_3^{-1} : C2616 $x =$ C2616 P2616* x

1,2,5 Additional P2616 captured by $\mathcal{C}$ $\mathcal{P}^*$ sub-pattern

3,4,6 Does not apply

7 Additional P2616 captured by (P2068 | P2616)* sub-pattern

f_4^{-1} : x S2616 $= x$ P2616* S2616

Additional P2616 captured by $\mathcal{P}^*$ $\mathcal{S}$ subpatterns.

f_5^{-1} : x P2616 $y = x$ P2616$^+$ y

1,3 Additional P2616 captured by $\mathcal{P}^*$ sub-patterns

2,4 Additional P2616 captured by $\mathcal{P}^*$ and P2616$^+$ sub-patterns

5,6,7 Additional P2616 captured by $\mathcal{P}^*$ and (P2068 | P2616)* sub-patterns

f_6^{-1} : x P2068^2 $y = x$ P2068$^{\geq 2}$ y

1,2 Additional P2068 captured by $\mathcal{P}^*$ sub-patterns

3,4 Does not apply

5 Additional P2068 captured by combination of $\mathcal{P}^*$ and (P2068 | P2616)* sub-patterns

6,7 Additional P2068 captured by (P2068 | P2616)* or $\mathcal{P}^*$ sub-patterns

f_7^{-1} : C2068 $x =$ C1945 P2068 x

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

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.

f_8^{-1} : x P2068^2 $y = x$ P2068 P2616 P2068 y

1 Produces arrangement which matches pattern 5

2 Additional P2616 captured by $\mathcal{P}^*$ sub-pattern

3,4 Does not apply

5 Additional P2616 captured by either $\mathcal{P}^*$ or (P2068 | P2616)* sub-patterns

6,7 Additional P2616 captured by $(\text{P2068} \mid \text{P2616})^*$ or $\mathcal{P}^*$ sub-patterns

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 eight reductions. □

B HTTP Persistent Connections

B.1 Equivalence Rules

1. x *proxy-10-cpc* y ▷ $\{x$ *proxy-10-cpc server-10-c*, *client-10-cpc* $y\}$
2. x *proxy-11-cpc* y ▷ $\{x$ *proxy-11-cpc server-11-c*, *client-11-c* $y\}$
3. x *proxy-10 proxy-10* y ▷ $\{x$ *proxy-10* $y\}$
4. x *proxy-10-c proxy-10 proxy-10-c* y ▷ $\{x$ *proxy-10-c proxy-10 server-10-c*, *client-10-c proxy-10 proxy-10-c* y, x *proxy-10-c* $y\}$
5. x *proxy-11 proxy-10 proxy-10-c* y ▷ $\{x$ *proxy-11 proxy-10-c* $y\}$
6. x *proxy-10 proxy-10-c proxy-11* y ▷ $\{x$ *proxy-10 proxy-11* $y\}$
7. x *proxy-11 proxy-10 proxy-11* y ▷ $\{x$ *proxy-11* $y\}$
8. x *proxy-11 proxy-10 proxy-11-c* y ▷ $\{x$ *proxy-11 proxy-11-c* $y\}$
9. x *proxy-10 proxy-11-c proxy-10* y ▷ x *proxy-10 proxy-11 proxy-10* y
10. x *proxy-10 proxy-11-c proxy-10-c* y ▷ x *proxy-10 proxy-11 proxy-10-c* y
11. x *proxy-10 proxy-11-c proxy-11* y ▷ x *proxy-10 proxy-11 proxy-11-c* y
12. x *proxy-10-c proxy-10-c* y ▷ $\{x$ *proxy-10-c* y, x *proxy-10-c server-10-c*, *client-10-c proxy-10-c* $y\}$
13. x *proxy-11 proxy-10-c proxy-11* y ▷ $\{x$ *proxy-11* $y\}$
14. x *proxy-11-c proxy-10-c proxy-11* y ▷ $\{x$ *proxy-11-c proxy-11* $y\}$
15. x *proxy-10-c proxy-11 proxy-10-c* y ▷ $\{x$ *proxy-10-c* $y\}$
16. x *proxy-10-c proxy-11 proxy-11-c* $y\ z$ ▷ $\{$*proxy-10-c proxy-11* $y\ z\}$ ($y\ z$ indicates $\mathcal{P}^+\mathcal{S}$)
17. x *proxy-11 proxy-10-c proxy-11-c* y ▷ $\{x$ *proxy-11 proxy-10-c server-11-c*, x *proxy-11 proxy-11-c* $y\}$
18. x *proxy-11-c proxy-10-c proxy-11-c* y ▷ $\{$*client-11-c proxy-10-c server-11-c*, x *proxy-11-c* $y\}$
19. x *proxy-10-c proxy-11-c proxy-10* y ▷ $\{x$ *proxy-10* y, *client-10-c server-11-c*$\}$
20. x *proxy-10-c proxy-11-c proxy-10-c* y ▷ $\{x$ *proxy-10-c proxy-11-c server-10-c*, *client-10-c proxy-11-c proxy-10-c* y, x *proxy-10-c* $y\}$
21. x *proxy-11-c proxy-11 proxy-10* y ▷ $\{x$ *proxy-11-c proxy-10* $y\}$
22. x *proxy-11-c proxy-11 proxy-10-c* y ▷ $\{x$ *proxy-11-c proxy-10-c* $y\}$

23. x *proxy-11 proxy-11* y $\rhd$ $\{x$ *proxy-11* $y\}$

24. x *proxy-11-c proxy-11 proxy-11-c* y $\rhd$ $\{x$ *proxy-11-c* $y\}$

25. x *proxy-11 proxy-11-c proxy-10* y $\rhd$ $\{x$ *proxy-11 proxy-10* $y\}$

26. x *proxy-11 proxy-11-c proxy-10-c* y $\rhd$ $\{x$ *proxy-11 proxy-10-c* $y\}$

27. x *proxy-11 proxy-11-c proxy-11* y $\rhd$ $\{x$ *proxy-11* $y\}$

28. x *proxy-11-c proxy-11-c* y $\rhd$ $\{x$ *proxy-11-c* $y\}$

29. x *proxy-11 proxy-10-c proxy-10 proxy-11* y $\rhd$ $\{x$ *proxy-11* $y,$ *client-10-c proxy-10 server-11*$\}$

C Web Intra-Cache Consistency

C.1 Reduction Rules

R_1 : x *proxy-plain* y $\rhd$ x y

(plain proxying has no effect upon caching)

R_2 : x *proxycache-plain proxycache-plain* y $\rhd$ x *proxycache-plain* y

(plain proxying has no incremental/marginal effect)

R_3 : x *proxy-scrubber proxy-scrubber* y $\rhd$ x *proxy-scrubber* y

(adjacent scrubbers have no incremental/marginal effect)

R_4 : x *proxy-scrubber proxycache-plain* y $\rhd$ x *proxycache-plain proxy-scrubber* y

(because *proxycache-plain*ignored BTC headers, the order of this pair doesn't matter, so we normalize it)

R_5 : *server-plain proxy-scrubber* x $\rhd$ *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* $\rhd$ x *client*

(a scrubbing proxy has no effect upon a cacheless client)

R_7 : x *proxycache-btc proxycache-btc* y $\rhd$ x *proxycache-btc* y

(successive BTC caches add no incremental value)

R_8 : x *proxycache-btcpush proxycache-btcpush* y $\rhd$ x *proxycache-btcpush* y

(as R_7)

R_9 : x *proxycache-btcpush proxycache-btc* y $\rhd$ x *proxycache-btcpush* y

(as R_7, and "push" effect passes through the *proxycache-btc*)

R_{10} : x *proxycache-btc proxycache-btcpush* y $\rhd$ x *proxycache-btcpush* y

(as R_7, and "push" effects are only downstream)

R_{11} : x *proxycache-btcpush proxycache-plain* y $\rhd$ x *proxycache-btcpush proxy-plain* y

("push" protocol disables downstream non-BTC caches)

R_{12} : x *proxy-scrubber proxycache-btc* $y \ \triangleright\ x$ *proxy-scrubber proxycache-plain* y

(no tokens reach the *proxycache-btc*, therefore it defaults to acting like a regular cache)

R_{13} : x *proxy-scrubber proxycache-btcpush* $y \ \triangleright\ x$ *proxy-scrubber proxycache-plain* y

(as R_{12})

R_{14} : x *proxycache-plain proxycache-btc* $y \ \triangleright\ x$ *proxycache-btc* y

(a BTC cache is unaffected by inconsistency introduced by the upstream *proxycache-plain*; if it receives tokens, it acts like a correct BTC cache, otherwise it behaves like a *proxycache-plain*.)

R_{15} : x *proxycache-plain proxycache-btcpush* $y \ \triangleright\ x$ *proxycache-btcpush* y

(as R_{14})

R_{16} : *server-plain proxycache-btc* $x \ \triangleright$ *server-plain proxycache-plain* x

(a plain server provides no BTC annotations, so *proxycache-btc* reverts to *proxycache-plain* behavior)

R_{17} : *server-plain proxycache-btcpush* $x \ \triangleright$ *server-plain proxycache-plain* x

(as R_{16})

R_{18} : *server-btc proxycache-btc* $x \ \triangleright$ *server-btc* x

(trivially follows from the correctness of BTC)

R_{19} : x *proxycache-btcpush client* $\triangleright\ x$ *proxycache-btc client*

(no agents separate the *proxycache-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)

C.2 Proof of Closure under Reductions

Lemma C.1. *The union of the patterns given in Theorem 3.3 correctly identify all members of $\mathcal{A}_\perp \cap \mathcal{A}_{\textbf{true}}$.*

Proof. The "brute-force" proof of this lemma is trivial, as $\mathcal{A}_\perp$ has four members:

server-plain client, *server-plain proxycache-plain client*, *server-btc client*, and *server-btc proxycache-plain client*.

Among these, *server-plain client* and *server-btc client* match the patterns which suggest they are members of $\mathcal{A}_{\textbf{true}}$; this agrees with a trivial analysis (there are no caches to introduce inconsistencies in either). Similarly, the other two arrangements must be members of $\mathcal{A}_{\textbf{false}}$; this agrees with a trivial analysis (there is nothing to prevent the *proxycache-plain* agent from introducing inconsistencies, which plain caches are able to do by their nature). So the pattern has correctly partitioned $\mathcal{A}_\perp$. □

Lemma C.2. *The set defined by the union of the three patterns given in Theorem 3.3 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 20 reductions R_1 through R_{20}, we examine its meaning with respect to members of each of the three success patterns stated in Theorem 3.3 as Equations 14, 15, and 16 (re-stated here as patterns 1, 2, and 3, respectively):

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

The 20 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* $|$ *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* $|$ *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

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

1 Does not apply

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

3 As under R_1

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* $|$ *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

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

1 Does not apply

2 Does not apply

3 As under R_3

R_5 : *server-plain 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 Does not apply

3 Does not apply

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

1 As under R_5

2 As 1

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

R_7 : x *proxycache-btc proxycache-btc* y $=$ x *proxycache-btc* y

1 Does not apply

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

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

R_8 : x *proxycache-btcpush proxycache-btcpush* y $=$ x *proxycache-btcpush* y

1 Does not apply

2 As under R_7, but for *proxycache-btcpush*

3 As under R_7, but for *proxycache-btcpush*

R_9 : x *proxycache-btcpush proxycache-btc* y $=$ x *proxycache-btcpush* y

1 Does not apply

2 As under R_7

3 As under R_7

R_{10} : x *proxycache-btc proxycache-btcpush* y $=$ x *proxycache-btcpush* y

1 Does not apply

2 As under R_8

3 As under R_8

R_{11} : x *proxycache-btcpush proxycache-plain* y $=$ x *proxycache-btcpush proxy-plain* y

1 Does not apply

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

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

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

1 Does not apply

2 Does not apply

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

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

1 Does not apply

2 Does not apply

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

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

1 Does not apply

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

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

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

1 Does not apply

2 As under R_{14}, but for *proxycache-btcpush*

3 As under R_{14}, but for *proxycache-btcpush*

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

1 Does not apply

2 Does not apply

3 Does not apply

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

1 Does not apply

2 Does not apply

3 Does not apply

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

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

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

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

R_{19} : x *proxycache-btcpush client* $=$ x *proxycache-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 *proxycache-btc* and *proxycache-btcpush* would be substituted within $\mathcal{P}^{*}$ term (no effect upon match)

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 Does not apply

3 Does not apply

Thus, the sets $\mathcal{A}_{\textbf{false}}$ and $\mathcal{A}_{\textbf{true}}$ are each closed under the reductions R_1 through R_{20}. $\square$