Modelling Enterprise System Protocols and Trace Conformance

Cameron Hine, Jean-Guy Schneider, Jun Han
Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218, Hawthorn, VIC 3122, AUSTRALIA
{chine,jschneider,jhan}@swin.edu.au

Steve Versteeg
CA Labs
Level 2, 380 St. Kilda Rd
Melbourne, VIC 3004, AUSTRALIA
steven.versteeg@ca.com

Abstract—Distributed enterprise systems, which are comprised of multiple software systems, typically communicate using application-layer protocols. Being able to independently test each system’s conformance to the protocol specification is important to ensure the overall robustness and reliability of the distributed system. While there exists many modelling languages for network layer protocols, very few tools exist for modelling valid message sequences specifically for the application layer. In order to address this issue, we propose a concise formal modelling syntax for application-layer protocols, with clearly defined semantics. A key feature of our protocol model is dynamic extension, which is necessary for the concise modelling of subsidiary concurrent operations. We demonstrate the power of our modelling language by compactly specifying the valid message sequences of two common application-layer protocols. Trace conformance is defined for the model along with a prototype implementation demonstrating the practical utility of our modelling framework.

Keywords—enterprise systems; protocol modelling; trace conformance; dynamic extension

I. INTRODUCTION

In a distributed system, communicating pairs (or end-points) are able to interact with one another over a network channel by agreeing to and faithfully implementing an appropriate protocol. At the enterprise level of abstraction, end-points transmit messages of different types across a network channel that may be asynchronous, but generally ensures in-order delivery. An enterprise protocol defines the acceptable order of message types which can be transmitted by both parties of an interaction over a network channel.

However, the open nature of distributed systems implies that developers of particular end-point systems cannot know the internal implementation details of all the other end-points with which their system may communicate. In fact, the only knowledge which can be obtained of potential interaction partners is of their observable behaviour; the messages they transmit on the network channel throughout an interaction.

Responsible end-point developers thoroughly test their systems using some combination of transparent and opaque box techniques. Moreover, responsible developers will attempt to verify their systems interactions with end-points that it may communicate with, once deployed. In particular, developers want to ensure that firstly, the transmissions sent by their system adhere to the protocol and secondly, that transmissions made by other end-points adhere to the protocol as well.

A significant portion of the desired interaction verification can be conducted offline. A trace describing the messages sent between two end-points on a network channel can be compared with what is defined as being valid in the corresponding protocol specification. However, in order to check the validity of the interaction from a particular end-point’s perspective, the trace must be collected in the appropriate location, and the protocol must be specified from that end-point’s perspective. For example, if the end-point being developed is some kind of server, then the trace is collected at the server end, preserving the order in which messages are transmitted and received. The protocol is specified in a way that the valid transmissions and receptions are defined for the server, corresponding to the trace collection point.

Protocol conformance testing [1] is the task of verifying a protocol implementation with respect to the rules set down by the protocol’s specification. Trace conformance is a particular aspect of protocol conformance testing which uses a formal specification of the protocol and a trace of an interaction in order to reveal whether the trace conforms to the protocol or not. In this work, we are particularly interested in trace conformance from the perspective of an individual participant of an interaction.

In order to check an implementation’s conformance, a precise and unambiguous specification of the valid interactions of the protocol is required. This specification can then be used as a basis for a communicating end-point’s required behaviour; all transmissions of the end-point must adhere to those defined in the specification for the implementation to conform. The enterprise systems we are interested in, typically communicate using application-layer protocols such as LDAP [2], HTTP [3], or SNMP [4]. Although ASN.1 [5] is commonly used to formally denote data structure, encoding and decoding of transmittable messages, details on message ordering are predominantly specified informally in Request for Comments (RFCs). In order to enable the verification of application-layer protocol traces, we introduce a formal protocol model specifically designed to facilitate the transformation of informal, text-
based message sequencing specifications into concise formal representations, appropriate for trace conformance testing. The introduction of a new formalism is required so that complex behaviour patterns exhibited by application-layer protocols can be modelled succinctly and in sufficient detail. In particular, the novel concepts of dynamic protocol extension and contraction, included in the new formalism, enable concise representation of complex communication patterns exhibited by application-layer protocols.

The formal specification of a protocol defined by our model is used as a basis for testing trace conformance. We define message evaluation rules which, given an instance of a protocol model specification, along with a particular message trace, determine whether the trace conforms to the protocol or not. The message evaluation rules can be used to notify enterprise end-point developers of potential faults in their implementation of the protocol, or of invalid behaviour by a communication partner.

The rest of this paper is organised as follows: Section II introduces the proposed protocol model, along with two example specifications, LDAP and BitTorrent, illustrating the applicability of the protocol model for different kinds of application-layer protocols. This is followed by the description of the evaluation rules for the protocol model in Section III, which also includes a brief discussion of a prototype implementation. Section IV provides a discussion of related work. Conclusions and directions for future work are given in Section V.

II. A MODEL FOR PROTOCOL SPECIFICATIONS

In this section, we propose an abstract protocol model to enable the concise and formal specification of application layer communication patterns. We also illustrate the model using the Lightweight Directory Access Protocol (LDAP) [2] and BitTorrent Peer Protocol [6] as examples.

A. Protocol Model Specification

The core idea of the protocol model is that two or more communication partners in an enterprise system exchange sequences of messages (or message traces). In order for the communication to be valid, the corresponding message trace must adhere to the rules defined in the corresponding protocol specification.

In our protocol model, we abstract this communication twofold. First, we define the set of all valid message types of a given protocol, denoted by \( T \), and associate a unique message type, denoted by \( t \in T \), to every message. Second, we specify the valid message traces of a protocol from the perspective of a single end-point. An end-point can either receive or transmit messages, but for the purpose of our modelling and analysis, the identity of the communication partners of a given end-point are irrelevant. Therefore, every exchanged message has a well-defined direction, either input or output, denoted by \(?\) or \(!\), respectively. As a consequence, every message \( m \) in the protocol model is denoted by a pair defining both, the direction as well as the type of this message.

For example, in the LDAP server specification given in Figure 2, \( ?\text{UnbindRq} \) denotes the receipt of an unbind request message, whereas \( !\text{DisconNot} \) denotes the transmission of a disconnect notification.

The abstract syntax of the protocol model is given in Figure 1. At the top-level, a protocol model specification \( S \) is defined as either a declaration \( D \) or a protocol specification \( P \). A declaration contains a non-empty sequence of variable declarations which bind a (locally defined) protocol specification \( P \) to a variable \( V \in \mathcal{V} \). We require the set \( \mathcal{V} \) of variable names to define equality and inequality.

A protocol specification \( P \) may specify the composition (or product) of two protocols, denoted by \( P \times P \), an extension of a protocol with another protocol, denoted by \( P \mathcal{P} \), an interaction, denoted by \( I \), a variable \( V \), or inaction, denoted by \( 0 \). An interaction \( I \) is either a choice between two interactions, denoted by \( I \mathcal{I} \), a standard interaction, denoted by \( m \mathcal{P} \) or a contractive interaction, denoted by \( m \mathcal{P} \). As discussed above, a message \( m \) is a pair consisting of a direction \( o \in \{?,!\} \) and a type \( t \in T \). Similar to \( \mathcal{V} \), we require equality and inequality to be defined for \( T \).

In protocol model specifications, (i) protocol extension takes precedence over product composition, and (ii) protocol extensions, product compositions, and choice are evaluated from left to right (e.g., \( P_1 [ P_2 ] [ P_3 ] \) is an extension of \( P_1 [ P_2 ] \) with \( P_3 \)). Round brackets may be used to group expressions in order to enhance readability and/or overcome the default precedence rules.

| \( S \) | Specification |
| \( D \) | Multi Declaration |
| \( V = P \) | Single Declaration |
| \( I \) | Choice |

Figure 1: Protocol Model Abstract Syntax.
The underlying idea of the protocol model is to specify all valid messages of a protocol specification at a specific point in time. This is expressed using the syntactical elements standard interaction and contractive interaction which define expected message types, respectively. The set of valid messages for the next interaction is determined by the actual message that was communicated. Therefore, both standard interaction and contractive interaction include a continuation, that is, a protocol specification to be used as the basis for testing the next interaction.

The specific composition element used, combined with (i) the continuation of a matching base interaction and (ii) the type of this interaction (standard or contractive), fully determines the set of valid interactions for the next interaction. The main difference between the two forms of base interaction is that in case of a contractive interaction, all extensions to a currently active base protocol are to be terminated (i.e. the valid message traces defined by these extensions become invalid), whereas in the standard form, all extension remain intact.

The syntactical elements product, extension, and choice allow us to express various compositions of protocol specifications and interactions, as discussed in more detail below. Choice between interactions allows a protocol specification to define multiple valid transmissions and/or receptions. If one of the interactions in a choice matches a given message, then this message is considered to be valid, and the protocol model specification uses the continuation of the matching interaction as the basis for testing the next interaction. All other choices are discarded. Unsuccessful matches in all of the choices result in a failure.

The protocol model defines two types of protocol specification compositions: product composition and extension composition. The product composition of two protocol specifications allows for two protocols to be combined in such a way that message traces can be interleaved. Therefore, a message \( m \) that is valid for either \( P_1 \) or \( P_2 \) in \( P_1 \ast P_2 \), is considered to be valid for the product composition, and the corresponding matching protocol specification will evolve, leaving the other protocol specification unchanged. If \( m \) is valid for both, \( P_1 \) and \( P_2 \) at the same time, only one of the protocol specifications evolves, but not the other. As a result, a successful match in one protocol specification does not change the valid interactions of the other.

Protocol extension is particularly useful in specifying temporary extensions to a base set of interactions. As an example, consider the search functionality of an LDAP server given in Figure 2. Searching is orthogonal to the underlying base functionality. However, any pending search operations must be immediately terminated if either an unbind or bind request is received by the server, or a disconnect notification is emitted. In case of a bind request, a bind response is emitted and the LDAP server resets itself. In the other two cases, the LDAP server will not accept any further interactions; it evolves into the inaction 0. Note that this behaviour cannot be specified using product composition alone.

A variable declaration \( V = P \) assigns the protocol specification \( P \) to the variable \( V \). All variables defined for a given protocol model specification are visible “globally” and, as a consequence, all (locally defined) protocol specifications can refer to these variables. Thus, protocols may refer to one another by name and recurse as required. In the LDAP server specification given in Figure 2, the protocol specification Base refers to both, itself as well as the protocol specification Search.

The reader may note that the proposed protocol model does not allow for the specification of nested declarations. None of the client-server and peer-to-peer protocols we analysed for this work required nested declarations. Furthermore, nested declarations would have significantly increased the complexity of introducing closed protocol model specifications (cf. Section II-C) and the definition of the protocol evaluation rules given in Section III. However, nested declarations is a topic of future work and will be further discussed in Section V.

From a different perspective, a protocol specification \( P \) can be interpreted as a binary tree with the syntactical elements standard interaction, contractive interaction, inaction, and variable as leaf nodes, and product, extension, and choice as non-leaf nodes, respectively. A depth-first traversal of the tree\(^1\) allows us to check whether a message \( m \) is valid in \( P \) or not. A successful match of \( m \) at one of the leaf nodes will trigger a rewriting of this tree, and the resulting tree can then be used to check the next message of an interaction. The formal semantics of protocol evaluation given in Section III is based on this interpretation.

### B. Example Specifications

In this section, we will illustrate the expressive power of the proposed protocol model by applying it to both, the LDAP and BitTorrent application-layer protocols, and discuss the main observations.

\(^1\)During this traversal, variable nodes will have to be replaced by their respective protocol specification.
The client-type interactions. Informally, an LDAP server is a Lightweight Directory Access Protocol (LDAP) server whenever a new search request is received. A search request, on the other hand, can result in zero or more search result entries which match the search criteria received by the server. After zero or more result entries have been transmitted, the search completion is indicated by the transmission of a search result done message. To complicate matters, an LDAP server has the ability to service multiple search requests simultaneously in a single LDAP session. These searches may complete in an arbitrary order and may even terminate before completion, either due to a request by the client or the server.

An LDAP session is usually closed by the client issuing an unbind request. A server, however, may also close the session after transmitting a disconnect notification. Any outstanding searches will be terminated when the session is closed. This will also happen if a client rebinds to the server, possibly using different authentication credentials.

Figure 2 illustrates the specification of the LDAP server protocol using the proposed protocol model. To enhance readability, all occurrences of protocol variables are underlined. We specify the basic protocol functionality in \( \text{Base} \), the functionality of searching in \( \text{Search} \), and extend \( \text{Base} \) with \( \text{Search} \) whenever a new search request is received. Similarly, in order to enable the non-blocking of an LDAP server in the context of processing administrative and data modifying requests, the \( \text{Base} \) protocol is extended with protocol specifications encoding the appropriate responses. Finally, contractive interactions are used to terminate any pending operations when an unbind or bind request is received, or a disconnect notification is emitted by the server.

**BitTorrent**: BitTorrent [7] is a peer-to-peer (P2P) protocol supporting the scalable distribution of large data-files to a large number of interconnected nodes. To achieve this, the data-files intended for distribution are split into a number of smaller pieces. A peer interested in downloading a file joins an appropriate BitTorrent swarm and proceeds to (i) retrieve pieces from other peers which it does not currently have, and (ii) makes available to other peers pieces which it has already obtained. Eventually, a peer will succeed in downloading all individual pieces of the requested data-file and can merge them into a full copy of the original data-file.

The BitTorrent peer protocol defines the majority of the interaction required to exchange pieces between peers. Figure 3 defines a specification of this peer protocol, assuming that the initial handshake and optional bitfield message exchanges have already taken place. This specification defines a well-behaved BitTorrent peer; a peer whose transmissions are congruent with its view of channel states.

Interaction behaviour of a BitTorrent peer is defined as the product of three protocols: \( \text{Base} \), a local channel state (initially \( C_L \)), and a remote channel state (initially \( C_R \)). The use of a product enables the \( \text{Base} \) local and remote state protocols to operate independently of one another; interactions of one do not effect the valid interactions of either of the others.

A search request, on the other hand, can result in zero or more search result entries which match the search criteria received by the server. After zero or more result entries have been transmitted, the search completion is indicated by the transmission of a search result done message. To complicate matters, an LDAP server has the ability to service multiple search requests simultaneously in a single LDAP session. These searches may complete in an arbitrary order and may even terminate before completion, either due to a request by the client or the server.

An LDAP session is usually closed by the client issuing an unbind request. A server, however, may also close the session after transmitting a disconnect notification. Any outstanding searches will be terminated when the session is closed. This will also happen if a client rebinds to the server, possibly using different authentication credentials.

Figure 2 illustrates the specification of the LDAP server protocol using the proposed protocol model. To enhance readability, all occurrences of protocol variables are underlined. We specify the basic protocol functionality in \( \text{Base} \), the functionality of searching in \( \text{Search} \), and extend \( \text{Base} \) with \( \text{Search} \) whenever a new search request is received. Similarly, in order to enable the non-blocking of an LDAP server in the context of processing administrative and data modifying requests, the \( \text{Base} \) protocol is extended with protocol specifications encoding the appropriate responses. Finally, contractive interactions are used to terminate any pending operations when an unbind or bind request is received, or a disconnect notification is emitted by the server.

**BitTorrent**: BitTorrent [7] is a peer-to-peer (P2P) protocol supporting the scalable distribution of large data-files to a large number of interconnected nodes. To achieve this, the data-files intended for distribution are split into a number of smaller pieces. A peer interested in downloading a file joins an appropriate BitTorrent swarm and proceeds to (i) retrieve pieces from other peers which it does not currently have, and (ii) makes available to other peers pieces which it has already obtained. Eventually, a peer will succeed in downloading all individual pieces of the requested data-file and can merge them into a full copy of the original data-file.

The BitTorrent peer protocol defines the majority of the interaction required to exchange pieces between peers. Figure 3 defines a specification of this peer protocol, assuming that the initial handshake and optional bitfield message exchanges have already taken place. This specification defines a well-behaved BitTorrent peer; a peer whose transmissions are congruent with its view of channel states.

Interaction behaviour of a BitTorrent peer is defined as the product of three protocols: \( \text{Base} \), a local channel state (initially \( C_L \)), and a remote channel state (initially \( C_R \)). The use of a product enables the \( \text{Base} \) local and remote state protocols to operate independently of one another; interactions of one do not effect the valid interactions of either of the others.
and remote are denoted by the $L$ and $R$ subscripts. The combination of two states forms the logical conjunction of the two states. $C_L I_L$, for example, represents the local channel state of being unchoked and interested.

The BitTorrent peer protocol defined in Figure 3 is well-behaved. A peer will only send stateful transmissions which will result in a meaningful change of state. When a remote peer is choked, it may send an unchoke, but it will not re-send a choke as this is already the state of the channel. Similarly, when the peer is interested, it will not re-send interested, only not-interested. Furthermore, a peer adhering to the protocol specification in Figure 3 will only send requests for pieces when it is unchoked by a remote peer and has registered its interest. Moreover, cancel messages may only be sent after a piece request has been made.

Product composition of the proposed protocol model enables the succinct specification of the BitTorrent peer protocol. Without product composition, 16 unique and somewhat lengthy declarations are required to define the valid interactions for the various channel states. Writing such specifications is not only tedious, but also increases the possibility of errors.

C. Closed Protocol Model Specifications

As detailed in Figure 1, the abstract syntax of the proposed protocol model allows for the use of variables $V$ in protocol specifications $P$. For every variable used in a protocol specification $P$, we expect that a corresponding declaration occurs in its overarching protocol model specification $S$. Undefined variables may result in a protocol model specification to be ill-defined.
In order to avoid this problem, we define the set of free variables of a protocol model specification \( S \), written \( f_V(S) \), as shown in Figure 4. For the purpose of this work, we consider a protocol model specification \( S \) to be closed if it does not contain any free variables, that is, \( f_V(S) = \emptyset \). Protocol model specifications \( S \) containing free variables are open. For the remainder of this work, we will only consider closed protocol model specifications.

### III. Protocol Evaluation

The protocol model defined in Section II evolves over the course of an interaction through the process of message evaluation. In this section, we detail the meaning of message evaluation in the proposed protocol model. We also briefly describe a prototype trace conformance checker which we have constructed. Please, note that message evaluation is only defined for closed protocol model specifications, that is, protocol model specifications not containing any free variables.

#### A. Preliminaries

The main motivation of our work is to test the validity of a message trace with regards to a given protocol model specification. We define \( \mathcal{M} \) to be the set of all messages \( m \) of a protocol model specification. As discussed in Section II-A, every message \( m \) is a pair \((o, t)\) with a direction \( o \in \{?, !\} \) and a type \( t \in T \). We also define \( \mathcal{M} \) to be the set of all finite message traces that can be constructed using \( \mathcal{M} \). A message trace, denoted by \( \bar{m} \in \mathcal{M} \), is either the empty trace \( [] \) or a non-empty, finite sequence of messages \( [m_1, m_2, \ldots, m_i] \). Hence, we define

\[ \mathcal{M} = \{[]\} \cup \{[m_1, m_2, \ldots, m_i] : m_i \in \mathcal{M}, i \in \mathbb{N}^+\} \]

We write \( [m_1 : m^*] \) to denote a non-empty message trace starting with message \( m_1 \).

Furthermore, we define a lookup context \( \Lambda \) to deal with declarations for a protocol model specification. A lookup context records pairs of variables \( V \in V \) and their assigned protocol specification \( P \in \mathcal{P} \). A lookup context may be the empty context, denoted by \( \langle \rangle \), or an extension of an existing context with a declaration, denoted by \( \Lambda(V = P) \).

\[
\begin{align*}
\text{lookup}(V, \langle \rangle) &= \varepsilon \\
\text{lookup}(V, \Lambda(V = P)) &= P \\
\text{lookup}(V', \Lambda(V = P)) &= \text{lookup}(V', \Lambda) \quad \text{if } V \neq V'
\end{align*}
\]

The function \( \text{lookup} \) defined above uses a lookup context to find the protocol specification \( P \) defined in a declaration \( V = P \). Failure to find an assignment results in a lookup failure \( \varepsilon \), whereas success returns the corresponding assigned protocol specification. Lookup contexts are very similar to forms \([8], [9]\) as they allow for the definition of key/value pairs, overriding of existing bindings, and a lookup mechanism that always returns the most recent binding for a key.

The heart of our message evaluation semantics are rules for interaction specification evaluation and protocol specification evaluation, testing the validity of a message in a protocol or interaction specification, respectively (cf. Figure 5). Both sets of rules generate annotated protocol continuations. Annotations are used to encode the result of matching a message against the set of valid messages, whereas a continuation defines the protocol specification to be used as the basis for testing the validity of the next message of an interaction trace in case of a successful match. Due to the different semantics of standard and contractive interactions, annotations encode whether a message matched in a standard interaction, denoted by \( \checkmark \), or a contractive interaction, denoted by \( \triangleright \). A failed match is encoded as \( \varnothing \).

We define the set of protocol continuations \( \mathcal{P}^* \), the set of protocol annotations \( \mathcal{A} \), and the set of annotated protocol continuations \( \mathcal{C} \) as follows:

\[
\mathcal{P}^* = \mathcal{P} \cup \{\perp\} \quad \mathcal{A} = \{\checkmark, \triangleright, \varnothing\} \quad \mathcal{C} = \mathcal{A} \times \mathcal{P}^*
\]

The reader may note that, in order to enhance readability of message evaluation, we use the symbol \( \perp \) to indicate the continuation in case of an unsuccessful match. The functions \text{type} and \text{cont} extract the type annotation and continuation protocol specification encoded in an annotated protocol continuation \( c \in \mathcal{C} \), respectively.

#### B. Message Evaluation

The meaning of a protocol model specification is defined in terms of a number of message evaluation rules, summarised in Figure 5. In this section, we will discuss the evaluation rules for single messages; the rules for message trace evaluation are discussed in Section III-C.

**Interaction Specification Evaluation:** Rules (R12) to (R14) given in Figure 5 define the meaning of message evaluation for interaction specifications. We have to consider the three syntactic categories choice, standard interaction, and contractive interaction.

The rules for standard interaction and contractive interaction define the fundamental building blocks of the proposed message evaluation semantics as they are the only rules to test for concrete messages in a protocol model specification. The rule for standard interaction (R13) evaluates a message \( m \) in \( m'.P \). If \( m \) is equal to \( m' \), then \( m \) is considered to be a valid message in the given context, and the rule generates the annotated protocol continuation \( (\checkmark, P) \) to indicate a successful match at a standard interaction. We shall use the term “successful standard match” to characterise such a behaviour. If \( m \) and \( m' \) differ, then a failure is signalled. Similarly, a successful match in a contractive interaction (R14) will generate \( (\triangleright, P) \) if the message \( m \) matches the expected message \( m' \). We shall use the term “successful contractive match” in this context. In case of a choice (R12), a successful match is obtained if at least one of the two available branches signals a positive match; the other branch is discarded.
\[ D \otimes \tilde{m} = (D \vdash \langle \rangle) \otimes \tilde{m} \quad (R1) \]
\[ P \otimes \tilde{m} = (P, \langle \rangle) \otimes \tilde{m} \quad (R2) \]
\[ (V = P \text{ and } D) \vdash \Lambda = D \vdash \Lambda (V = P) \quad (R3) \]
\[ (V = P \text{ in } P') \vdash \Lambda = (P', \Lambda (V = P)) \quad (R4) \]
\[ (P, \Lambda) \otimes [\ ] = \checkmark \quad (R5) \]
\[ (P, \Lambda) \otimes [m : m^*] = \begin{cases} \phi & \text{if } \text{type}(P, \Lambda) = \phi \\ (\cont((P, \Lambda) \circ m), \Lambda) \otimes [m^*] & \text{otherwise} \end{cases} \quad (R6) \]
\[ (P_L \ast P_R, \Lambda) \circ m = \begin{cases} (\text{type}(P_L, \Lambda) \circ m), (\cont(P_L, \Lambda) \circ m) \ast P_R & \text{if } \text{type}(P_L, \Lambda) \circ m) \in \{\checkmark, \triangleright\} \\ (\text{type}(P_R, \Lambda) \circ m), P_L \ast (\cont((P_R, \Lambda) \circ m)) & \text{if } \text{type}(P_R, \Lambda) \circ m) \in \{\checkmark, \triangleright\} \\ (\phi, \bot) & \text{otherwise} \end{cases} \quad (R7) \]
\[ (P_L [P_R], \Lambda) \circ m = \begin{cases} (\checkmark, (\cont(P_L, \Lambda) \circ m) [P_R]) & \text{if } \text{type}(P_L, \Lambda) \circ m) = \checkmark \\ (\triangleright, (\cont(P_L, \Lambda) \circ m)) & \text{if } \text{type}(P_L, \Lambda) \circ m) = \triangleright \\ (\text{type}((P_R, \Lambda) \circ m), P_L [\cont((P_R, \Lambda) \circ m)]) & \text{if } \text{type}((P_R, \Lambda) \circ m) \in \{\checkmark, \triangleright\} \\ (\phi, \bot) & \text{otherwise} \end{cases} \quad (R8) \]
\[ (V, \Lambda) \circ m = (\text{lookup}(V, \Lambda), \Lambda) \circ m \quad (R9) \]
\[ (I, \Lambda) \circ m = I \ast m \quad (R10) \]
\[ (0, \Lambda) \circ m = (\phi, \bot) \quad (R11) \]
\[ (I_L + I_R) \ast m = \begin{cases} I_L \ast m & \text{if } \text{type}(I_L \ast m) \in \{\checkmark, \triangleright\} \\ I_R \ast m & \text{if } \text{type}(I_R \ast m) \in \{\checkmark, \triangleright\} \\ (\phi, \bot) & \text{otherwise} \end{cases} \quad (R12) \]
\[ m'.P \ast m = \begin{cases} (\checkmark, P) & \text{if } m' = m \\ (\phi, \bot) & \text{otherwise} \end{cases} \quad (R13) \]
\[ m'.\triangleright P \ast m = \begin{cases} (\triangleright, P) & \text{if } m' = m \\ (\phi, \bot) & \text{otherwise} \end{cases} \quad (R14) \]

Figure 5: Message Evaluation Rules.
Protocol Specification Evaluation: The meaning of message evaluation for protocol specifications are given by the rules (R7) to (R11), covering the corresponding five syntactic categories. Message evaluation at the protocol specification level requires a lookup context \( \Lambda \) in order to replace any free occurrences of variables by their respective declarations.

The rules for the syntactic categories \textit{inaction} and \textit{interaction} are straightforward. The \( 0 \) protocol specification considers every possible message as invalid and, as a consequence, message evaluation will always result in a failure \( \phi \) and invalid continuation \( \bot \) (R11). Message evaluation for interactions at a protocol specification level invokes an appropriate rule at the interaction specification level (R10).

(R9) defines the message evaluation for variables. In essence, a variable is looked up in the given lookup context \( \Lambda \) and the bound protocol specification is used to proceed with the evaluation process. Due to the fact that message evaluation is only defined for \textit{closed protocol model specifications}, no lookup error \( \varepsilon \) can occur.

Of more interest are the rules for \textit{product composition} and \textit{protocol extension}, respectively. Similar to the rule for choice, the message evaluation of a product composition succeeds if the message \( m \) is valid in either of its two branches. If this is the case, the continuation protocol is defined as the product composition of the continuation of the matching branch with the \textit{original} protocol specification of the non-matching branch (cases 1 and 2 of (R7)). The annotation of the successful match is preserved in this process. The remaining case deals with failed matches in both branches. (R7) guarantees that product composition is a symmetric operation. Thus, the two branches of a product composition can be swapped without changing the set of valid message traces.

The message evaluation rule for protocol extension is similar to the one for product composition. The main difference is that the two types of successful matches \( \sqrt{\cdot} \) and \( \triangleright \) in the left branch (\textit{i.e.} the \textit{base} protocol) are treated differently. In the first case, the continuation protocol generated by the successful match is extended with the right branch (\textit{i.e.} the \textit{extension} protocol). In the second case, however, the extension protocol is discarded, and only the continuation protocol generated by the successful match is used. Treating these two cases differently is the main reason why the message evaluation rules for interaction and protocol specifications generate \textit{annotated protocol continuations}; we need to be able to distinguish between the continuation generated by \( m.P \) and the one generated by \( m \triangleright P \). Please note that protocol extension is not a symmetric operation as exchanging the base protocol with the extension protocol generally leads to different valid message traces.

If a message \( m \) is matched by both branches in either a product composition, protocol extension, or choice, the corresponding rules do not specify which of the available options to use (\textit{i.e.} we have \textit{non-deterministic} behaviour). If non-determinism is not the desired behaviour for product composition, for example, then either of the first two cases in (R7) can be amended with an additional condition requiring that message evaluation in the other branch results in a failure \( (\phi, \bot) \). By doing so, preference will be given to either the left or the right branch, respectively. A similar modification can also be applied to (R8) and (R12) in order to resolve non-determinism.

C. Trace Conformance

(R1) to (R6) define the rules for \textit{message trace evaluation}. The heart of this process is defined by (R5) and (R6). We consider an empty message trace \( [] \) as being valid for \textit{any} protocol specification \( P \) and lookup context \( \Lambda \) (R5). In case of a non-empty message trace \( [m_1 : m^*] \), we first apply the appropriate rules for protocol specification evaluation to \( m_1 \) (R6). If this results in a failure, the evaluation process is terminated, and \( [m_1 : m^*] \) is considered to be invalid. Otherwise, we continue the process on the remainder \( m^* \) of the message trace until we either fully succeed (\textit{i.e.} (R5) is applicable) or obtain a failure. The recursive process always applies the continuation protocol of a successful match to evaluate the next message of a message trace (R6).

(R5) and (R6) guarantee that the evaluation of a valid message trace always results in \( \sqrt{\cdot} \), independent of whether the evaluation of individual messages resulted in a standard or contractive matches, respectively.

Message trace evaluation requires a lookup context \( \Lambda \) containing all declarations given in a protocol model specification. (R1), (R3), and (R4) define the rules to create the required lookup context and bootstrap the evaluation process. In case a protocol model specification contains no declarations, the empty context is used for lookup purposes and (R2) is applied. It is worth noting that a lookup context \( \Lambda \) is not modified once (R5) and (R6) become applicable.

D. Prototype Implementation

As proof of concept to validate our work, we have developed prototypes for both, a protocol model specification parser and a message trace conformance validator implementing the proposed protocol model and message evaluation rules, respectively. The corresponding implementations were written in Haskell [10] and use the Parsec library for parsing protocol model specifications. Haskell’s functional style and support for pattern matching allowed the implementation of message evaluation to flow naturally from the rules defined in Figure 5.
The message trace conformance validator was used to check the conformance of a variety of LDAP traces, some were collected from real LDAP implementations and others hand crafted. The results of the trace conformance tests were all as expected; both conforming and non-conforming message traces were identified. Due to a lack of time, we have not yet been able to extensively validate BitTorrent message traces.

IV. DISCUSSION AND RELATED WORK

Formal specification languages such as Estelle [11], LOTOS [12], and SDL [13], along with their associated tools, have found significant use in the telecommunications domain in order to aid in the design and verification of lower-level communication protocols. Designers of application layer protocols, however, have not yet embraced these techniques. Message ordering for protocols such as LDAP [2], HTTP [3], and SNMP [4], is specified informally and published as Request for Comments (RFC’s). This situation makes it necessary to translate these informal descriptions into more formal representation before trace conformance can be performed.

Rather than using an existing specification language such as LOTOS or SDL for the target of such transformations, we proposed a specialised protocol model specification language in Section II. Our expectation being that a specialised model would be better suited to concisely expressing the behaviour patterns of application-layer protocols. This expectation was justified since the combination of the novel concepts of protocol extension and contraction allowed us to concisely define subsidiary extensions to valid interactions; a pattern that occurs frequently in LDAP.

Of the formalisms designed to specify communication protocols, our model is closest in style to session types [14]. Similarly to session types, the proposed protocol model is intended to specify the sequence of interactions between two parties in terms of the types of messages being exchanged. Unlike session types, however, our protocol model is channel agnostic; there is no explicit tie between a given protocol and a channel. This means that, although the protocol model was originally intended to specify interactions between two parties, messages could in fact be sent and received from multiple sources. Consequently, the protocol model described in Section II can be said to be at a higher level of abstraction than other similar models.

The trace conformance conducted in this work can be classified as a local single-layer test method [1] used to verify the conformance of a single trace from the perspective of a local system under test. The message trace conformance rules defined in Section III-C are a special kind of conformance test which is always conclusive; the message trace either conforms to the given protocol or it does not. Addressing the question of appropriate test selection, where adequate traces are produced that will give reliable general protocol conformance testing results, is beyond the scope of this work.

Protocol conformance is an active concern in multi-agent systems. Similarly to our work, there are efforts to ensure that distributed systems whose internal structure cannot be known, interact according to expectation. Unlike our approach, logic based formulas [15] and deterministic finite automata [16] are used to model protocol specifications. Furthermore, Endriss et al. [16] model different levels of protocol conformance so that conformance of an agent may be defined as weak, exhaustive, or robust. Our work has not yet required conformance definitions of this complexity.

Yu et al. examine conformance for HTTP [17]. The problem domain of their work is very similar to ours. They use the results of their HTTP investigation to make general remarks about problems of testing application-layer protocol conformance. A significant difference between their and our work is in the choice of model for valid behaviour. Yu et al. use a tree and tabular combined notation (TTCN) [18] to model multiple test cases designed to check if an implementation adheres to a protocol specification. Our approach, on the other hand, uses a model of the protocol itself as the definition of conformance.

V. CONCLUSIONS AND FUTURE WORK

This work presents a formal model designed specifically for description of behaviour patterns exhibited by application layer protocols. The model defines a protocol in terms of the message types allowed to be sent and received by a particular participant at specific points of an interaction. A novel aspect of this model is protocol extension, which allows a base protocol to be extended with some additional orthogonal behaviour. An extension is a subsidiary of its base and can be later terminated by its base if so desired. We demonstrate extension along with other aspects of the protocol model in example specifications of LDAP and BitTorrent.

We demonstrate one utility of the protocol model by defining trace conformance. Given an instance of a protocol model and a trace of message types, trace conformance can evaluate whether the trace does or does not conform to the supplied protocol. As a proof of concept, we have developed a prototype protocol model specification parser and implementation of the trace conformance rules. This prototype can take arbitrary protocol model specifications and message traces; given that a model of a protocol exists, conformance can be readily checked for protocols other than LDAP and BitTorrent.

A limitation of the protocol model is its inability to define nested declarations. Unfortunately, the increase in complexity caused by nested declarations meant that we did not have the time to include them in this version of the protocol model. If required in the future, nested declarations could be achieved through careful application
of \( \alpha \)-conversion techniques or hierarchical lookup contexts, respectively.

The environments in which enterprise software systems are deployed are notoriously complex. A significant problem facing enterprise system developers is replicating these complex environments for testing purposes. In previous work [19], we took steps towards addressing this problem through modelling and emulation. We modelled protocols using finite state machines which, unfortunately, made modelling of an arbitrary number of outstanding LDAP searches impossible. The protocol model presented here has overcome the limitation of the previous model through the concepts of extension and contraction. Future work will include incorporating the improved protocol model and trace conformance ideas into an enterprise emulation environment. This will enable an emulation environment to know what transmissions are valid at each point in an interaction, and to detect invalid communication from a system under test.

**ACKNOWLEDGEMENT**

We would like to thank Markus Lumpe for his input in defining the message trace evaluation semantics, Bernie Pope for his helpful discussions, CA for their ongoing financial support, and finally, the anonymous reviewers for their comments on the draft manuscript.

**REFERENCES**


