USING EVENT-BASED TRANSLATION TO SUPPORT DYNAMIC PROTOCOL EVOLUTION

by

NATHAN D. RYAN
B.A., University of Minnesota at Morris, 1998
M.S., University of Colorado at Boulder, 2000

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Doctor of Philosophy
Department of Computer Science
2004

This thesis entitled:
Using Event-Based Translation to Support Dynamic Protocol Evolution
written by Nathan D. Ryan
has been approved for the Department of Computer Science

_____
(Alexander L. Wolf)

_____
(Dennis Heimbigner)

_____
Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Ryan, Nathan D. (Ph.D., Computer Science)
Using Event-Based Translation to Support Dynamic Protocol Evolution
Thesis directed by Professor Alexander L. Wolf

All systems built from distributed components involve the use of one or more protocols for inter-component communication. Whether these protocols are based on a broadly used "standard" or are specially designed for a particular application, they are likely to evolve. The goal of the work described here is to contribute techniques that can support *protocol evolution*. We are concerned not with how or why a protocol might evolve, or even whether that evolution is in some sense correct. Rather, our concern is with making it possible for applications to accommodate protocol changes dynamically. Our approach is based on a method for isolating the syntactic details of a protocol from the semantic concepts manipulated within components. Protocol syntax is formally specified in terms of tokens, message structures, and message sequences. Event-based translation techniques are used in a novel way to present to the application the semantic concepts embodied by these syntactic elements.

For Elisa, and the memories that are yet to come.

Acknowledgements

First and foremost I would like to thank my advisor, Alexander Wolf, not only for acting as my advisor in this endeavor, but also for his patience and understanding regarding the quirks of my (sometimes difficult) personality. His clear-mindedness acted as the compass by which this work was guided, his steady hand often grounding ideas that might otherwise have simply flown away. Without his encouragement and assistance, the work presented here would not have come to fruition.

I would also like to thank my fellow members of SERL, the research group of which I have been a part from the very beginning of my studies at the University of Colorado. In particular, I offer warm thanks to Dennis Heimbigner for his thoughtful and objective input as well as his friendship. Also, thanks to the members who greeted me upon my arrival and made me feel welcome: André van der Hoek, for the regular games of pool and for introducing me to so many things I would not have discovered on my own; Antonio Carzaniga, for his unique perspectives and healthy dose of skepticism; Rick Hall, for teaching me to say "blast it all" to skepticism in so many euphemisms; Judy Stafford, for the introduction to Boulder so long ago, and for the ready laughs and encouragement; Dave Reese, for the mild-mannered conversation and accommodation; Thorna Humphries, for the endless optimism and the wonderful Thanksgiving dinner when I was yet homeless; and Ken Anderson, who began his career as I arrived and whose enthusiasm was a star during dark times, even if I never told him.

A special thanks also to those who I have met along the way, who have lifted my spirits beyond the academic: Marco Castaldi and Mauro Caporuscio, for introducing me to life; Naveed Arshad, for introducing me to another life; Rick Osborne, for teaching me how to fly (literally), and for his ready ear; Dana Soukup, for her belief in the impossible; The Forum 407 (for certainly there are that many of you), for maintaining what sanity is left to me; John Giacomoni, for the movies, assistance, and tech talk; and, especially, Laura Vidal, for everything.

Some historic gratitude is due, for there are those without whom I would not have arrived at the start of this grand adventure. Thank you to the dynamic duo of Scott Lewandowski and Nic McPhee, who are appreciated more than they know. Thank you to Heidi Lindgren for the years of friendship and the deep caring that sometimes came at your own expense. Thank you to Jonah Fetzer, for keeping it real. Thank you to Mike Rentz and Leslie Clapper-Rentz, for being an anchor even when their own seas were tumultuous, and for granting my life an epic relationship.

Thanks to my parents and my sisters, who maintained a staunch belief in my abilities even when I did not have the courage. Thanks to my ready and ample supply of familial role models. Especially, thanks to My Precious, who is immortalized by the ways in which I pass her influence to others; I'm sure that I have made her proud.

Newton once remarked, "If I have seen farther than others, it is because I have stood on the shoulders of giants." If such is true, then the distance to which I can see is limited only by my own nearsightedness, and is no fault of the army upon whose shoulders I stand. To all those who have not been mentioned for reasons of brevity, you are in my heart.

CONTENTS

FIGURES

1. Introduction

The goal of this work is to enable dynamic communication protocols, which is to say, to allow a distributed application to continue functioning even when the communication protocols used by its distributed components have changed. For our purposes, a protocol for distributed communication is considered to be a form of addressed, application-level message passing. In this work we are not concerned with network-level protocol issues, such as routing and forwarding.

Traditionally, an application uses a protocol by having close ties to the *syntactic details* of the protocol. Here, syntax refers to the structure of messages and the rules governing the coordination of messages passed among multiple components. Thus, a rule stating that a "response" message must follow a "request" message is considered as much a part of the protocol syntax as is the textual format of each "response" and "request" message.

The embedding of the syntactic details of a protocol into the code of an application means that changes to the protocol might force alterations to the application in order to accommodate those changes. Even small changes or additions introduced into a protocol—such as the redefinition of a token, an appended message structure, or a supplemental acknowledgement message for coordination—can have

drastic implications for an application, possibly forcing the application to be redesigned, modified, and rebuilt according to the new protocol specification. Instead, we would prefer the application be concerned with the *semantic concepts* encapsulated by the protocol, rather than its syntactical details.

A semantic concept (or simply "concept") represents an element of the data or behavioral logic of a component interaction. A common example is the concept of "date", which can have many syntactic manifestations. Another example is the concept of "request" in an HTTP protocol interaction; the fact that different versions of the protocol represent this concept in different syntactic forms should be largely irrelevant to an HTTP client or server component.

Typically, however, there is nothing available to decouple semantic concepts from syntactic details on behalf of an application, so the application either must itself extract the concepts from the syntax or must rely solely on the syntax as a representation of concepts. Either way, the behavior of the application is tightly coupled with the syntactic details of the protocol. Therefore, if the protocol is altered—whether the syntactic change reflects a true concept change—the application cannot continue.

Clearly, the issue of syntactic and semantic separation has been long recognized in various communities, and a variety of technologies have been proposed to address the problem. For example, the database community has explored the problem of multi-database integration, and has offered the technology of what are called *mediators* as a solution [Wie92]. Mediators serve to perform automatic translation of data and data requests, either to and from a common universal schema,

or on a database-to-database pair-wise basis. In the software architecture community, the latest attempt at solving the problem comes in the form of the *connector* [CBB+03], which is an architectural element whose purpose is to encapsulate inter-component communication. The idea is that replacing one connector with another to effect a protocol change should be possible, even at run time.

The distinction between syntactic detail and semantic concept is not always easy to discern. To some extent, it is a matter of exercising the proper discipline in the design of the application. Our goal is to provide tools and techniques that encourage the designer to make this distinction explicit. The incentive that we offer is the ability to develop components that can transparently accommodate certain kinds of protocol changes.

Our approach is based on a novel use of *event-based translation*. The idea behind event-based translation is to correlate "events" with syntactic structures. The parsing half of this technique is demonstrated to some degree by the XML SAX parser [MB02], which generates events as the syntax of an XML document is parsed. Each event captures a concept represented by the XML syntax of the document, and presents it to the application in the form of a method call belonging to a handler object. Inspired by the XML SAX parser, we expand this basic idea to achieve the separation of protocol syntax from protocol concepts, as represented in Figure 1.

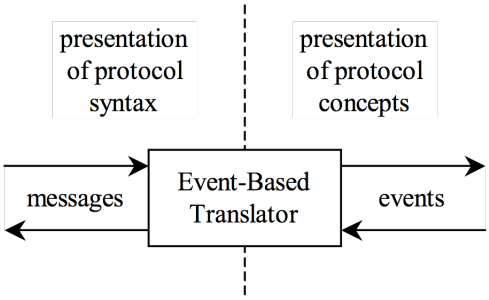

Figure 1: Separating protocol syntax from protocol concepts

An event or sequence of events can embody an abstraction of one or more syntactic elements, specifically the semantic concept or concepts associated with those elements. This means that concepts of the protocol are captured by the events produced by the translation system and given to a component, isolating the component from the syntax of the protocol. The event-based translation technique can be generalized and formalized so that an event-based translator can be constructed on demand. If the protocol specification changes, a new event-based translator can be constructed to replace the old one, and the application can continue to receive the events in which it is interested.

We stress that there are limits to the nature of changes that can be accommodated by our approach; those that involve deep semantic changes will obviously require changes to the components themselves. Moreover, there are some development-time and run-time costs to adopting the approach. Our experience indicates that there is a sufficiently rich set of changes that can be handled by our approach and, perhaps more importantly, that these changes are found in the real-world evolution of popular protocols. For example, HTTP 1.1 introduced a new message, the “status 100” continue-to-wait message, a change to the protocol that

can, through our technique, be made transparent to an HTTP 1.0 client. Regarding run-time cost, we are faced with a classic tradeoff of performance (both programmer and program) against flexibility. We have interposed a level of indirection in the form of an event-based translator, which requires some programmer effort to develop and some computer effort to execute. Further evaluation of the intellectual and computational overheads involved in this tradeoff will be pursued in future work.

The remainder of this chapter will be devoted to an overview of the approach and a breakdown of the aspects that comprise the approach. In the next chapter we discuss related work. Following that we present the approach in more detail, first discussing the principles and practice of what we require for protocol specification, and then discussing the requirement for realizing an implementing system. We then overview a prototype of the event-based translation system before discussing its use with respect to two protocols, HTTP and SIENA, and implications to the various Weblogs interfaces and XML RPC.

1.1. General Approach

The general approach we have developed to support dynamic protocol evolution consists of two main elements: a four-part specification of protocol syntax (discussed in Chapter 4), and an event-based technology for translating from the specified syntax into semantic concepts (and vice versa, the issues of which will be discussed in Chapter 5). The four-part specification serves as a means to conveniently modularize the different aspects of a protocol that might evolve, while the event-based translation system, whose behavior is driven by the syntax specification, presents semantic concepts as abstract events. The relationship between syntactic

elements and semantic concepts is currently established through a simple mapping from the elements to a set of concept names, although one could easily imagine using an arbitrarily sophisticated ontological scheme instead.

1.1.1. Specification of Protocol Syntax

We treat a protocol generically as a form of application-level message passing. Most well known protocols for distributed communication have a precise definition for the structure of a "message", and at least an informal description of what constitutes "passing". If we assume that messages can be viewed structurally as documents (a reasonable assumption), then we can use document description techniques to specify the structure of messages as tokens, as well as to specify the compositions of tokens (that is, the format of a message).

However, this is clearly not the complete definition of a protocol. Message specifications say nothing with regard to the rules of coordination among messages, so additional information is necessary. We refer to these rules as an *interaction specification*. Note that we require several specifications for each interaction of a protocol, one specification for each role a component may assume while using the protocol. Further, these specifications may include information that affects the way in which messages are delivered, such as transport bindings, reconstruction of partial messages, special timeouts, and the like. These last aspects of a protocol are in a sense orthogonal to the main goal of our work and are not currently addressed, but their influence on a protocol must be acknowledged.

Thus, we have a four-part syntax specification. Figure 2 depicts the "uses" relationship among these elements.

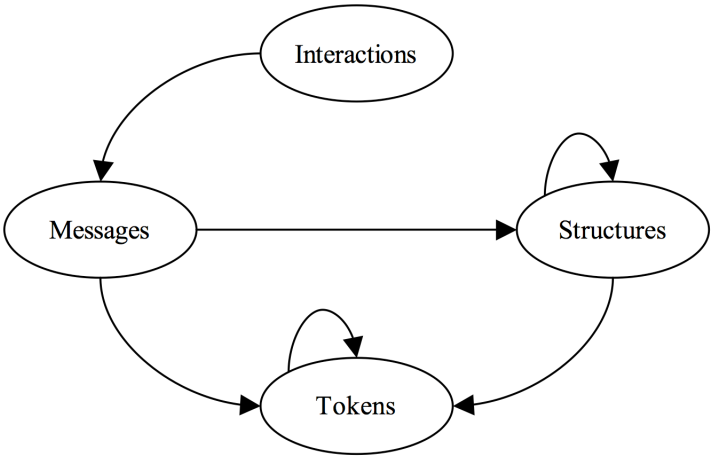

Figure 2: The four parts of protocol syntax specification

Protocol syntax is only one of the two primary aspects of protocol specification, however, the other being the association of semantic concepts with the syntactic elements. Figure 3 replicates Figure 2, adding the association of the appropriate concept types with each part of the syntax.

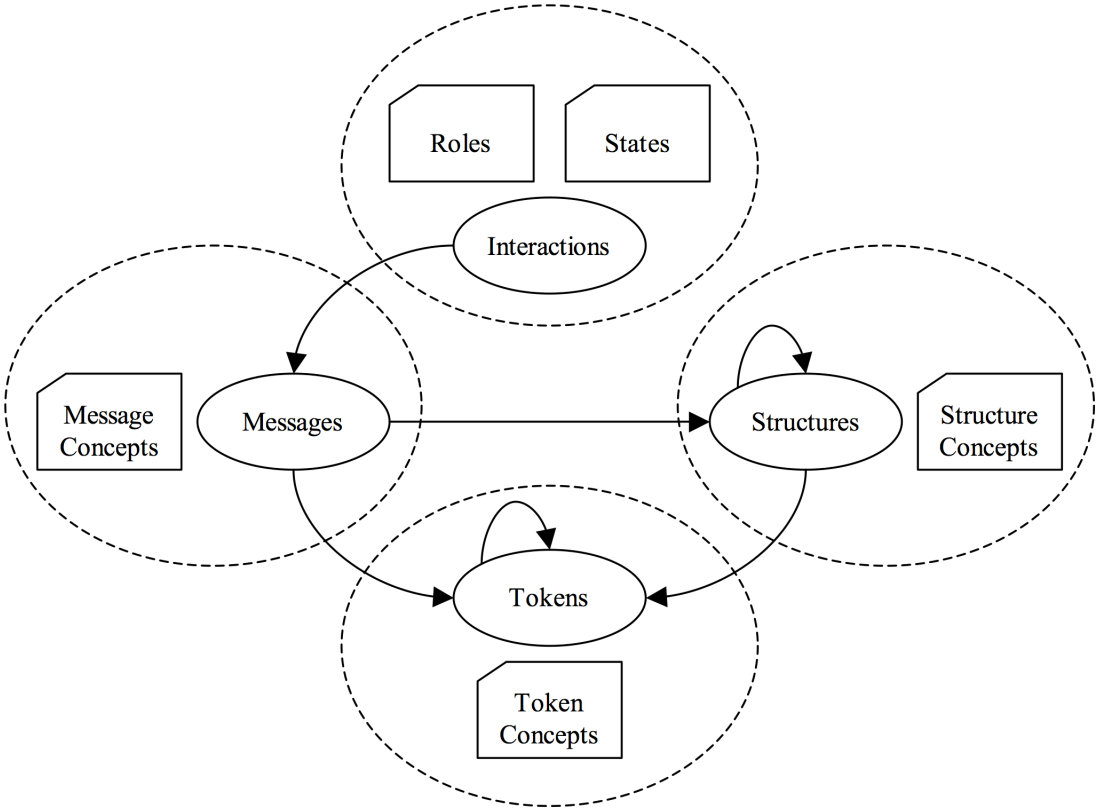

Figure 3: Concept association with the four parts of protocol syntax

As Figure 3 shows, messages, structures, and tokens can be associated with concept names belonging to a corresponding class. Interactions are, in addition to being defined by messages, partially defined by role concepts and state concepts; roles are conceptual names for the various components participating in the interaction, and states are conceptual names for the states of the interaction from the perspective of one or more roles.

Given this specification framework, a description language must be chosen (or developed) to satisfy each of the four parts of protocol syntax, simultaneously allowing for concept association. A suitable, although not necessarily distinct, description language should be assigned to each part, where we take “suitable” to

imply at least the following four criteria: (1) sufficient expressiveness, (2) compact representation, (3) ease of (visual) manipulation, and (4) isolation of changes.

A balance must be kept among these criteria when assigning description languages to each of the four parts. Clearly, important conflicts exist among the criteria. Additionally, conflicts may arise among the description languages chosen for each part of the specification framework, due to the dependencies among the parts (noted in Figure 2). For example, since the specification of messages depends on the specification of tokens, the presentation of tokens must be compatible with their use in the specification of messages.

In this work, we have adopted basic grammar languages for messages, structures, and tokens, since the expressive power required by structures and messages specifications is generally context free, while the expressive power required by token specifications is generally regular. The overall structure of a message, however, does require some context-sensitive specification such as, for example, the common situation that arises when one message structure element is used to indicate how many of another message structure element appears in the message (e.g., the value of the "Content-Length" HTTP header indicates the length of the included entity body). Thus, for message structures, we use description languages inspired by those found in common compiler-compiler tools such as Lex/YACC [LMB92], and more specifically, JavaCC[1].

---

[1] https://javacc.dev.java.net/
JavaCC was developed by S. Viswanadha and S. Sankar.

For the interaction specifications we currently use a simple language of state machines with guarded transitions. Note, however, that the choice of language features to capture information describing the way in which messages are delivered is problematic, since issues such as the recomposition of messages from possibly out-of-order communications and transport bindings do not immediately lend themselves to a simple state-machine language. Thus, in the future, we may add special language features for this aspect of the specification.

1.1.2. Isolation from Protocol Syntax

The primary elements of our approach consist of the four-part *syntax specification*, described above, an association of *semantic concepts* to syntactic elements, and an event-based *translation system* that, in turn, provides a coordination framework in the form of a *negotiator* responsible for negotiating protocols and a *generator* responsible for generating/managing event-based *translators* to operate on specific versions of the evolving protocol. A particular distributed component interacts with the translation system through events, which are either derived from or composed into protocol messages. Figure 4 shows these elements in relation, illustrating how the interposition technique allows the distributed application component to be isolated from the syntactic details of the protocol. (The coordination among the components of this figure is discussed in detail in Chapter 3.)

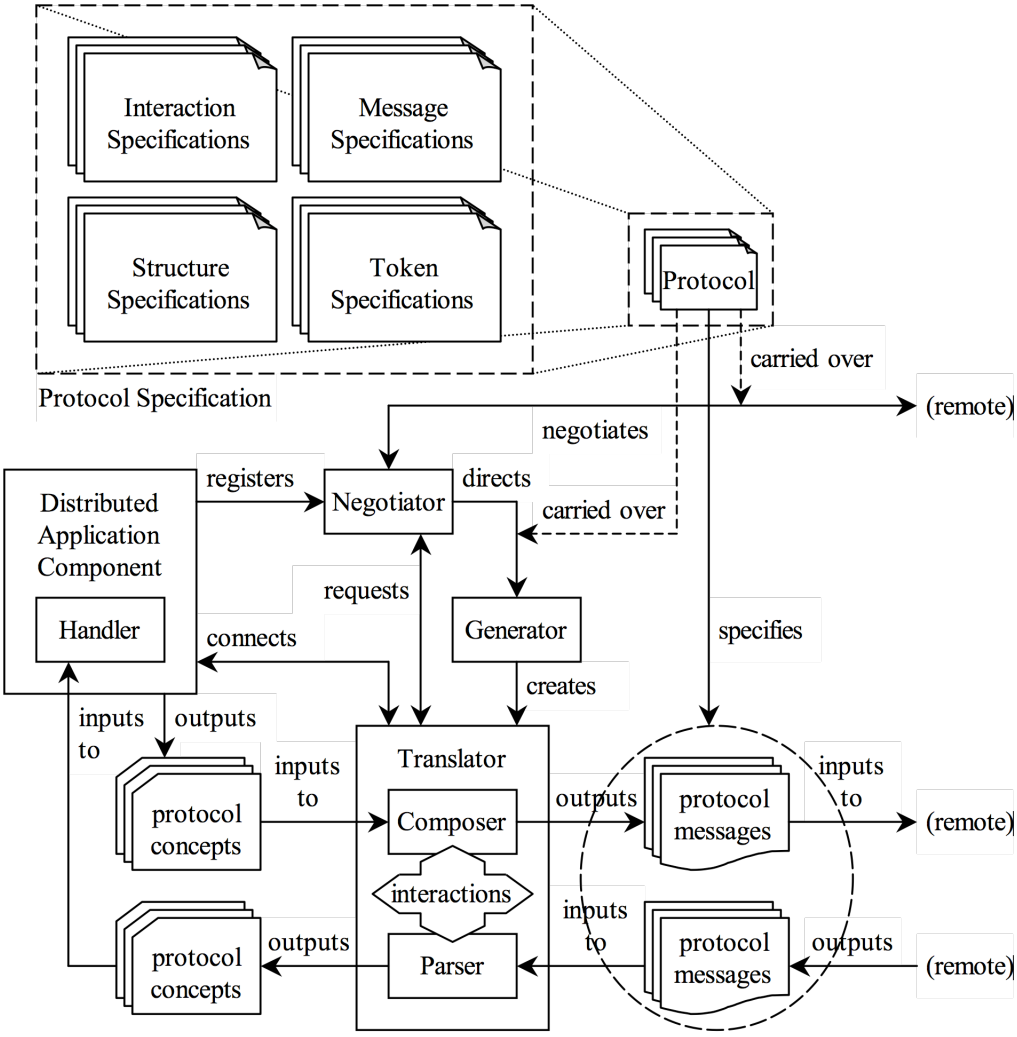

Figure 4: Isolating a component from protocol syntax

The negotiator element of the event-based translation system typically collaborates with a remote negotiator (not shown) for a corresponding remote component (also not shown) when the two distributed components first initiate communication. It is then the responsibility of the negotiators to determine which protocol will be used, given the preferences and capabilities of the relevant applications. If the negotiator determines that a protocol will be used for which a

translator does not exist, it will direct the generator element to create one, obtaining the protocol specification from the other negotiator, if necessary.

The protocol specification is used as input to the generator element of the event-based translation system. The generator creates a translator that recognizes the specified protocol. The translator is responsible for deconstructing protocol messages via the parser, constructing protocol messages via the composer, and maintaining state for each protocol interaction.

Protocol messages are input to the parser from a source component (possibly the composer of a translator used by the remote distributed component) engaged in the communication. The parser deconstructs the message and outputs the corresponding semantic concepts as a series of events. As the events are generated, they are delivered to an *event handler* in the distributed component. Once delivered, it is up to the component to decide how to interpret the events.

For the return path, the component gives a set of concepts to the composer, which uses those concepts to construct a legal message (or legal messages) of the protocol. That message is then sent to a target component, which in the general case may be different from the source of any protocol messages originally passed to the parser.

Notice that the distributed application component must cooperate with the translator, in the sense that it incorporates an event handler capable of processing the events generated by the parser, as well as being able to give appropriately encapsulated protocol concepts to the composer. (Section 3.3 will introduce some alternate configurations that allow for distributed-application components that were

not designed for an event-based translation system.) In the absence of such syntactic/semantic separation, the component can conceptually provide its own tailored translator, directly incorporated into its functionality, for each protocol it understands.

To summarize, the primary aspects of the system are interaction-state tracking, message parsing, and message composition. (The bulk of Chapter 5 will be devoted to issues involving state tracking, parsing, and composing.) Translation relates these primary aspects together. Generation is the automatic construction and instantiation of a translator for a protocol, given a specification for the protocol. Negotiation is the determination of which version of which protocol is best suited for communication among the components of a distributed application.

These aspects rely on several principles, the most notable of which are formal protocol specification and the formalization of event-based parsing/composing. (Chapter 4 is devoted to issues of protocol specification, with respect to the requirements imposed by the event-based translation system.) Other principles, no less important but having been so extensively studied that they need little more than a rudimentary review, relate to formal languages, particularly with respect to deterministic finite automata and compiler theory.

1.2. Hypotheses, Evaluation, and Contributions

We first hypothesize that, in addition to the simplistic examples presented throughout this work, realistic protocols can be formally specified using a four-part protocol syntax specification, and, further, that such a specification helps to isolate changes within the protocol syntax from one version to another. These hypotheses

will be evaluated by the example specification of protocols used for the case studies; some aspects of protocols that *cannot* be easily represented will also be presented.

We also propose that, given a formal specification of a protocol, we can automatically construct components that are capable of fulfilling the requirements of event-based translation (*viz.*, interaction-state tracking, message parsing, and message composition); this requires the formalization of event-based parsing, something that we will provide in relation to the protocol specification languages that we have developed. We will validate these propositions by providing the techniques that accomplish these tasks.

As these methodologies are explained, we will argue that they, in conjunction with a coordination framework, facilitate the swapping of protocols at runtime, thus enabling dynamic protocol evolution. As a demonstration, the communication between entities that understand different versions of a protocol will also be explored by the case studies.

The resulting technique for dynamic protocol evolution is the primary contribution of this work. We also will explain how this technique has the potential to expand into other areas of component interaction (not merely the communication among components of a distributed application), something to be explored as future work. Additional contributions include the refinement of event-based parsing and an innovative process for the construction of a string from a grammar.

The next chapter will attempt to cover previous work related to the aspects and principles of event-based translation, as well as discuss some competing methodologies to our approach. The following three chapters will examine the finer

techniques, requirements, and issues of the theory behind our methodology. Chapter 6 and Chapter 7 will present the practical side, first discussing a prototype system, and then discussing some experience with existing protocols. The final chapter will conclude by reviewing the contributions and limitations of this work and opportunities for future work.

2. Related Work

The ideas presented by this work can be seen as a particular approach to developing a dynamic connector among components, where the interface between a component and a connector is modeled as events. In that sense it is related to work in the area of software architecture and the treatment of connectors as "first-class objects". Garlan et al. point out that "allowing complex connectors provides a single home where one can talk about the semantics", also noting that "[one] could attach a single description of the protocol of interaction to the complex connector" [CBB+03, page 113]. The same viewpoint is adopted here. We perceive the protocol to be the semantics of an arbitrary connector that is capable of being dynamically swapped with another connector. Figure 5 shows this viewpoint, where the black boxes attached to each component indicate the coupling of the connector with the component.

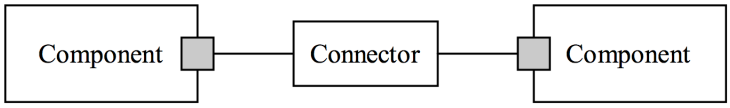

Figure 5: Connector coupled with components

The mechanics of realizing such a vision, however, are not straightforward. Garlan et al. note, "even simple interactions, such as message sending or procedure calls, become complex in their own right in most distributed settings" [CBB+03, page 113]. To make the separation between component and connector possible, the connector must be able to present some sort of coupling interface to its attendant components.

Proposals for such couplers already exist. Jones, Romanovsky, and Welch [JRW02] developed an approach based on the notion of *integrators*. Integrators are used as the coupling between connector and component, and are attached to each component with "glue code", thus providing a known interface to clients. Unfortunately, this approach requires a separate integrator for each interface/protocol pair, similar to early program translation methods, which required translation networks that included a unique translation between each pair of programming languages. For programming languages, this problem was simplified by the introduction of abstract languages and algebras [Bac59] [Bur65]. However, for the method proposed by Jones, Romanovsky, and Welch, which focuses on a fault tolerance mechanism with multi-layered exception handling, each integrator (essentially the translator) must be specifically designed with a particular protocol and interface in mind; no unified intermediary is proposed.

An important approach is taken in the architecture description language Wright [AG97]. Wright connectors use *roles* to define the behaviors involved in an interaction, where the connectors themselves describe the behavior of the connection. The roles are coupled with *ports* associated with the components. The language used to describe the behavior of the interactions is a variant of CSP [Hoa85], a powerful (largely symbolic) protocol description language that is accompanied by a wide variety of validation tools.

Although the Wright model is general enough to be used in almost any situation, it still relies on the notion of a protocol-specific coupling between components and connectors, as embodied in roles, ports, and the "glue" that binds them. If a component cannot match the specific image of the protocol that is presented to it through the definitions of the various roles and ports, the connector cannot be coupled with the component even if the roles and ports conceptually present the same information to which the component is accustomed. Figure 6 depicts the potential for such a mismatch.

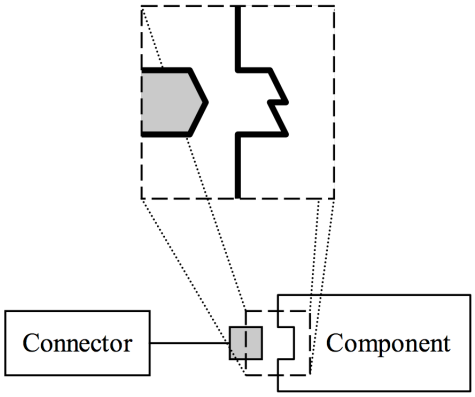

Figure 6: Coupling mismatch

The approach described here uses a mechanism that loosens the requirements for the coupling between a component and a connector, in a sense providing a "universal plug" for a connector to be coupled with a component. Conceivably, any connector can then be coupled to any component, regardless of the semantics associated with the connector. Therein lies the primary difference between most existing techniques and the one described here. Current realizations of the component/connector approach define mechanisms that rely on a strong specification of the interface provided by the connector and require that the component be strictly matched with the interface, whereas our approach provides a technique that generalizes the coupling. Of course, this flexibility comes at a price, and that price is the ability to predict *a priori* the compatibility of the connector and component simply by examining the interface. We must appeal to some outside mechanism to guarantee, or at least check, on compatibility.

This apparent conflict in approaches can be resolved if we treat a mechanism such as that found in Wright as a *design* aid and treat our approach as an

implementation aid. There is no reason why a strict model of the coupling interface of component and connector could not be used in conjunction with a faithful, albeit looser implementation of that interface.

The C2 architectural style and its supporting infrastructure take a similar approach to strictness and flexibility [TMA+96]. Strictness is achieved through an external mechanism that is part of the C2 development environment. At run time, flexibility is achieved through a common event bus to which arbitrary components can connect, disconnect, and reconnect. C2 differs from our approach in that it has adopted a universal protocol for all component interactions, which happens to be based on events. In contrast, we are trying to capture a variety of specific protocols, and use events at a different conceptual level, namely the boundary between the protocol and the components.

2.1. Protocol Specification

Given that our focus is on communication protocols for distributed systems, it is important to briefly explain more broadly the relationship of our work to that of the many tools and techniques available to perform protocol specification, such as the general-purpose specification languages CSP [Hoa85], LOTOS [EVD89], and Esterel [BS91] [CCD+94], as well as the architecture description languages Darwin [MDE+95] [PC96], Rapide [Luc96] [LKA+95], and Wright [AG97]. As we explain in Chapter 4, our approach relies heavily on the formal description of a protocol. Theoretically, we could use any of the popular protocol specification languages. The choice among them reduces to convenience, such as availability of tools and simplicity of conception, more than any other factor. For example, Esterel is

a full-fledged programming language for describing signal processing in reactive systems. As such, it appears to be too heavy weight for our purpose. We simply need a means by which to specify the format of messages and the coordination rules among messages. For the former, traditional grammar specification languages suffice, and for the latter, a state machine specification language is suitable; Chapter 4 will further enumerate our requirements.

Ultimately, despite the availability of these related formalisms, it was more convenient to develop our own set of specification languages (defined in Appendix B). The description of interactions requires some finite state machine specification, explained below, and the description of protocol messages requires an annotated grammar. Message description is consequently related to attribute grammars [Alb91] and variants (such as conditional attribute grammars [Boy96]). However, although we do require attributes of a sort, they are not used for semantic checking but rather as an aid for passing conceptual information to applications and for the piecemeal construction of valid protocol messages.

2.2. Interaction-State Tracking

The activity of interaction-state tracking amounts to a trace through a finite state machine, given that we define interactions as describable by a regular language. (For a detailed discussion on the nature of finite automata, see any textbook on computer science theory; for example, Sipser [Sip97]). The transactions proposed by Lustman can serve as a good example [Lus94]. Even Czarnecki and

Eisenecker [CE00, page 360] note, in relation to the CAPE environment[2] based on Draco [Nei84], "[A] communication protocol can be nicely defined using a finite state automata."

This is not entirely true; one could easily concoct a protocol of token passing that is not regular. However, we have been unable to discover any protocols whose interactions cannot be easily described by a finite-state automaton (although some, particularly in the arena of security/authentication, bear closer inspection [Bel03], which may influence future work), despite the fact that some specification languages used to describe protocol interactions are capable of describing much more complicated communication (such as the CCS calculus [Mil82] and the CSP process algebra [Hoa85], both of which can describe communication among concurrent processes in the larger sense). Therefore, although communication can be more complex (and languages for describing such complexity exist), we will restrict ourselves to interactions describable by a regular language, as has been done with the CAPE system.

2.3. Parsing

We turn now to parsing, an excellent survey of which can be found in [GJ90]. Of course, the foundations of parsing were laid decades ago, and the field has advanced and is now well understood, at least at the basic level. We will not attempt to revisit the history of, say, recursive-descent parsers or depth-first (Unger) parsers, except to note that we require some parser technology and, since we restrict the grammar of a protocol message to LL(1) (as explained in Chapter 4), a top-down

---

[2] http://www.bayfronttechnologies.com/

parser seems to be the most logical choice. A number of tools (some of which will be briefly discussed in Chapter 6) are available to assist in this process.

The additional consideration is that the parser must produce events, rather than construct, say, a syntax tree. For that we have looked most closely at the XML SAX parser [MB02] in its three versions. However, it is specific to the XML syntax (which uses a restricted grammar), rather than to the syntax of documents specified by a DTD or Schema. The XML SAX parser serves as a good model, but we require a more general form of event-based parsing.

2.4. Composing

Composition provides the channel from a component into the protocol, which explains our use of double-headed arrows in Figure 1. It also explains our use of the more general term “translation”, rather than just “parsing”. There are two phases to the composition of a protocol message: the *construction* of the tree that represents the message, and the *composition* of the message from the tree.

The former provides us with another reason to look beyond the XML SAX parser, since SAX utilities are incapable of performing the complement of parsing; however, XML can serve as a model here, as well. Various tool sets, specifically ones that implement the DOM interface [LLW+04], allow one to perform both the construction and the composition of an XML document, but only after the elements supplied to construct the abstract syntax tree of the document can be validated against the relevant DTD or schema. These validation methodologies are the basis for a construction technique suitable for our purpose here. Again, however, the XML DOM interface is specific to the XML syntax, rather than the language described by the

corresponding DTD/Schema. Instead, we will describe a technique that utilizes a data structure similar to a rule tree as used by Cracknell and Downton [CD98].

The composition of a DOM tree into an XML document follows a similar pattern to the behavior of another class of tools referred to as “unparsers”, such as those suggested for program source transformation (e.g., [Lov77] [PE88] [BD99]). Unparsers typically perform some traversal over an existing syntax tree, in order to effect textual formatting, as with “pretty printers”, or serialization of the tree. The unparser, then, only performs the second half of the composition activity for a specific grammar. The composition of a protocol message as described by this work is similar to that of an unparser, but may construct *ad hoc* subtrees during the process, something not performed by an unparser.

Another field that requires the construction and composition of strings from a grammar is that of natural language generation (NLG). Here, those composers (referred to as “generators” in the NLG literature, not to be confused with “translator generators” in this work) that rely on a grammar (versus those that rely on heuristics) tend to use small, specialized grammars so that the resulting string—formed in the final stage of composition, known as “surface realizations”—is more consistent with understandable language and can more easily incorporate conversation history [DSE98]. However, the strategies for the composition of natural language sentences (which, given grammars of any significant size, tend toward statistical heuristics) are not deterministic enough to adopt for the composition of protocol messages.

2.5. Translation

We define translation as the coordination among the activities of interaction-state tracking, parsing, and composing. Consequently, the relevance of other work to translation can be as wide or as narrow as one prefers, depending on the perspective. Every application that uses a protocol, for example, must provide some means to coordinate these activities according to the definition of the protocol in question, and as such are (in a sense) relevant to the more general translation activity described by this work. On the other hand, given that no prior work seems to have focused on the generalization of these techniques in order to provide a mechanism that is capable of dynamically swapping these protocols as they evolve, no prior work is relevant.

Rather than list the mechanics of every protocol imaginable as influencing what amounts to the relatively simple coordination of outgoing and incoming messages, we will instead take for granted that every application-level protocol, signaling mechanism, and interface has contributed in its own small way and concentrate on the generation of the translator itself.

2.6. Translator Generation

The generation of a translator based on the specification of a protocol is more than the execution of a compiler compiler; it requires the automatic creation and instantiation of a suite of logical architectural components whose coordination is not only predetermined but also automatic. Further, the generated components must themselves be generators of a sort, capable of performing (possibly oblique [CE00, page 343]) data transformations.

We thus surface into the larger context of generative programming, as we will be producing families of complex translators based on evolutions of a protocol specification. Generative programming corresponds to the real-world notion of an automated factory assembly line. In the words of Czarnecki and Eisenecker [CE00]:

> Generative Programming (GP) is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediary or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

Although one could parade a nearly unending list of utilities that perform or support generative programming, we will restrict ourselves to mentioning those that seem to have the most bearing on or parallel to this work.

We begin with the notion of dynamic, on-demand compilation, as with a Java JIT (Just-In-Time) compiler [LY99]. Normally, a Java Virtual Machine executes a program via interpretation of the corresponding bytecode. However, under certain conditions [CFM+97] [KCS+00] [Sch03], the performance of the JVM can be improved by compiling the bytecode into the native machine code and executing the machine code. While such optimization is not the focus of this work, the technique of the on-demand compilation of a higher-level specification is an essential ingredient.

Another aspect of the generation activity is assumed by the Draco paradigm, which provides a larger environment in which one defines a domain-specific generation activity [Nei84]. Draco and the derived system CAPE, already mentioned in Section 2.2, will be again revisited in the last section of this chapter. Other generation methodologies include GenVoca [BST+94], which uses stacked layers of

abstraction to perform iterative refinement[3], and Intentional Programming [Sim95], which introduces the idea of *active source*.

Of course, this generation occurs within a logical component framework. One cannot escape mentioning frameworks/architectures such a JavaBeans [KA98], CORBA [Sie00], and more recently .NET [Pla03], which tend to sprout—or to some extent include, in the case of Interface Definition Language (IDL) compilers and the Dynamic Invocation Interface (DII) of CORBA [LW98]—their own framework-specific generation tools, such as the Eclipse Modeling Framework[4] for Java and the Object Model Generator[5] for .NET.

However, we are not concerned with the specific technology used for generation or the specific framework in which the generation occurs, only the results of the generation itself. Any of these methodologies and frameworks could potentially provide a suitable infrastructure to implement the coordination framework and the translators that enable dynamic protocol evolution, as described by this work; we mention them as foundational material that at the very least provide a model for the generation activity. This brings us to the generation of the individual components of a translator, rather than generation in general.

The generation of an interaction state tracker from a specification is trivial, given that we restrict an interaction to what can be described by a deterministic finite automaton (DFA). Generators specifically for recognizing DFA and their transitions

---

[3] An excellent overview of GenVoca is provided by Czarnecki and Eisenecker [CE00].
[4] http://www.eclipse.org/emf/
[5] http://spaz.ice.org/code/ObjectModelGenerator/

abound, including AutoFSM[6], FSMGenerator[7], and NunniFSMGen[8], though the construction of a generator is relatively simplistic. The protocol compiler developed by Castelluccia, Dabbous, and O'Malley [CDO97] is another example of a compiler that produces automata, though from an abstract protocol specification written in Esterel. This work is promising, not only because it demonstrates the feasibility of such a technique, but also because it indicates that automatically generated translators can be efficient as well.

Although we make note of these tools (each capable of generating the interaction-state tracker component of a translator), their relation to this work is as evidence of the solved nature of this aspect of translator generation in both theory and practice. Though any of them could potentially be used, for our prototype we chose JavaCC for all aspects of generation, in order to help unifying the notation of the specification languages.

The generation of a parser from a specification is nearly as old as third-generation programming languages; a myriad of tools can be found to accommodate a variety of features (some of which will be discussed in Chapter 6). Aho, Sethi, and Ullman provide one of the quintessential resources on the theory and techniques of parser generation [ASU86]. However, given that we explicitly restrict the expressiveness of protocol messages to LL(1) (somewhat for practical reasons), it should be explicitly noted that the origins of this class of languages and the related "bottom-up" class of languages, LR(*k*), are discussed by Knuth in [Knu65].

---

[6] http://autogen.sourceforge.net/autofsm.html
[7] http://fsmgenerator.sourceforge.net/
[8] http://nunnifsmgen.nunnisoft.ch/en/home.jsp

On the other hand, the history behind composer generation from a specification is far less grounded than that of parser generation. The generation of a component capable of deterministically (versus statistically, as is typically the case with NLG) constructing a string conforming to the language defined by a specification seems restricted to the subject of unparsers, which, recall, commonly take the form of textual "pretty printers" [BV96] [Cam88]). The Eli translator suite [GLH+92], for example, includes a utility (Idem[9]) for unparser generation (both textual and structural). However, no generation system seems capable of generating the full message composition technique for dynamic protocol evolution, which requires not only the automatic composition of a string conforming to a grammar, but also the automatic construction of a tree from which that string is derived. As the focus of unparsers is essentially the composition of a string from a syntax tree, grammar-specific unparser generation is related to translator generation, in the sense that unparser generators are capable of producing the "second half" of the behavior required by a composer.

2.7. Protocol Negotiation

The coordination framework, responsible for determining which of a particular version of a protocol to use, necessitates some negotiation among the communicating parties. While we do not specify what form this negotiation takes (in general, a determination was obligatory for the development of a prototype), we can point to other areas where similar negotiation is used.

---

[9] http://eli-project.sourceforge.net/elionline4.4/idem_4.html

One area is that of security protocols. The Secure Sockets Layer (SSL) protocol, for example, specifies the negotiation of security parameters for further message communication between two entities using a "handshake" [FKK96] [WS96]. The Internet Key Exchange (IKE) protocol instead uses a two-phase negotiation for the establishment of a connection [HC98]. IKE is part of the IP Security (IPSec) framework that also includes the Internet Security Association and Key Management Protocol (ISAKMP), another means of negotiating security parameters [MSS+98] [SHR+02].

Another area is that of hardware signal negotiation, as with a RTS/CTS (request-to-send/clear-to-send) hardware handshake that can be used by, for example, the RS-232 interface standard for serial devices [TIA97] or the newer IEEE 802.11 Medium Access Control (MAC) layer for wireless networks [IEEE99] [XGB02] in order to negotiate data transfer rate.

Still other areas exist that are at least tangentially related to protocol negotiation, from universal personal computing (e.g., [THZ+03]) to network load balancing (e.g., [KHB02]). However, our intention in this work is not to provide or even suggest the means by which protocol negotiation must take place, only to recognize its potential with distributed applications whose components may evolve independently.

2.8. Competing Approaches

Several systems already mentioned, namely the Rapide modeling process [LKA+95], the Wright architectural model [AG97] (which uses CSP [Hoa85]), and the CAPE tool suite [Bay93] by Bayfront, are all well suited for

the description and analysis of protocol interactions. The first two are more targeted at software architectures rather than protocols, however; Rapide focuses on the modeling of system behavior as partially ordered sets of events (or "posets"), and Wright focuses on explicit connector types. CAPE, on the other hand, is specifically intended for protocol development; the Protocol Definition Language (PDL) is used to describe a state machine that represents a protocol (interaction), and this PDL description is used to automatically generate documentation, simulation code, and implementation code for the state machine.

Each of these systems is unable to satisfy our vision of dynamically evolvable protocols, however, because each lacks a mechanism for specifying (declaratively) the format of messages passed among application components during an interaction. Message specification, in addition to interaction specification, is required for the generation of an intermediary (i.e., a translator) that decouples an application component from the syntax of a protocol. Consequently, although these three systems (and others) are suitable (perhaps even preferable, given their more comprehensive analysis, e.g., [Luc96]) for describing communication among distributed entities, they automate only the communication, lacking the descriptive techniques for what is communicated, and are thus insufficient for our needs.

On the other side of the spectrum we have the software renovation factory proposed by Brunekreef and Diertens [BD99], which automates global source transformations according to grammar specifications. Given a grammar specification, a transformation factory is generated, which includes both parsers and unparsers for layered transformations. (The generation methodology is described by van den Brand

and Visser [BV96], and further by van den Brand, Sellink, and Verhoef [BSV00]). While this work is promising for both the decomposition and composition activities of protocol transformation, it does not incorporate the communication aspects necessary for interactions nor does it address the construction of source (message) trees.

A more comprehensive approach is taken by Reussner with the CoCoNut tool suite [Reu03], which performs protocol adaptation by means of a type system for the dynamic composition of software components; component interfaces are defined not only according to the way in which an individual method is called but also the way in which sequences of methods can be called. By mapping messages to method calls and interactions to sequences of method calls, one could thus specify a full protocol and use the dynamic composition mechanism to accomplish the same goal as dynamic protocol evolution; however, the format of messages is in this way limited, not to mention that our reliance on message syntax (in this case, the number, type, and order of method arguments) is not necessarily removed.

At a lower level, a systems approach is taken by the x-kernel protocol framework, as described by Druschel et al. [DAP+93]. The x-kernel protocol framework is capable of the full specification of a protocol via the Morpheus special-purpose programming language [AP93]. However, the dynamic nature of protocols described by Morpheus within the x-kernel system are aimed at maximizing the efficiency of network-level protocols, and have little to do with the isolation of target applications from the syntax of those protocols.

3. Coordination Framework

Coordination among the elements shown in Figure 4 of Chapter 1 is nontrivial, particularly given that two (or more) instantiations of the figure will be operating conjointly. Here we give an overview of this coordination and how it aids us in providing the service of event-based translation, thus enabling dynamic protocol evolution. This section is intended to provide a basic knowledge of the overall system for the clarity of later discussion.

We begin with the simple hypothetical of two (or more) components of a distributed application, which must use some protocol or set of protocols to communicate effectively. This conceptual situation is shown in Figure 7.

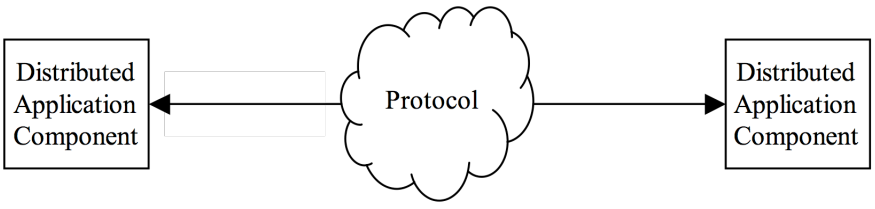

Figure 7: Distributed application communication

All parties involved must agree upon the protocol in use, however. This can cause difficulties for distributed applications in situations in which a relevant protocol may evolve. For example, if a single entity is not in control of the distribution of all

components of such an application, components may be upgraded independently to different versions of the protocol, causing inconsistencies in the absence of full backward compatibility. This is not a contrived example but rather a reflection of the real-world situations that result from common, publicly available protocols—such as HTTP [FGM+99], the SSH connection protocol [YL04], and a host of others—that seem to continuously undergo review, revision, and extension.

In order to rectify this potential conflict, we remove the burden of supporting a specific protocol syntax from the application, placing such responsibility in intervening middleware, as shown in Figure 8.

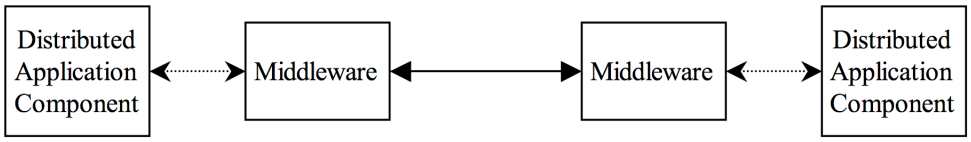

Figure 8: Middleware assumption of communication

Now, the communication of protocol messages (shown as a solid arrow), and by implication the understanding of the protocol syntax, is assumed by the middleware components, whereas each distributed application component can be concerned only with a lowest-level translation of the syntax into conceptual elements (shown as dotted arrows), as communication from (and to) the corresponding middleware component. Given that the responsibility for understanding the protocol syntax has been removed from the application components, the middleware components can now communicate using *any* protocol that encapsulates a sufficient subset of the concepts required by the application components, regardless of the protocol for which they were designed. This scenario affords us the ability to connect,

by middleware proxy, the two distributed application components even if, say, a different version of some common protocol was envisioned for each. (Note that we assume the middleware components will communicate using the same version of the same protocol, regardless of their concept presentations.)

If each application component is unaware of the (different) intended protocol version of the other component, the intervening middleware may not be available, as the requirement of its presence is unpredictable. We would then prefer the middleware to be dynamically generated, so the exponential number of possibilities in rectifying different protocol versions can be handled on demand. Figure 9 shows the addition of a middleware generator to each side of the implicit machine boundary (crossed by the solid, double-ended arrow representing the communication of protocol messages).

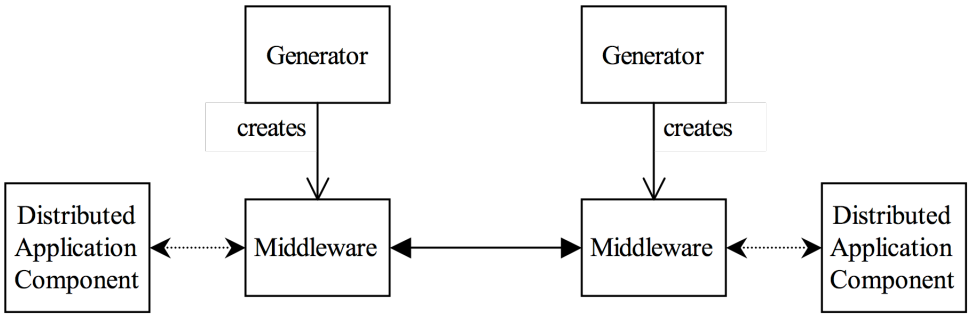

Figure 9: Middleware generation

Despite our dynamic generation we have merely pushed the understanding of the protocol syntax to another layer, rather than removing it entirely (which is unrealistic). Furthermore, the generation of the middleware itself requires some formal specification, including a specification of the protocol syntax in order for the generated component to understand it. Thus, we have a formal specification of the

syntax (which includes the association of syntactic element to concepts) of the protocol version to be used to each of the generators, as depicted by Figure 10.

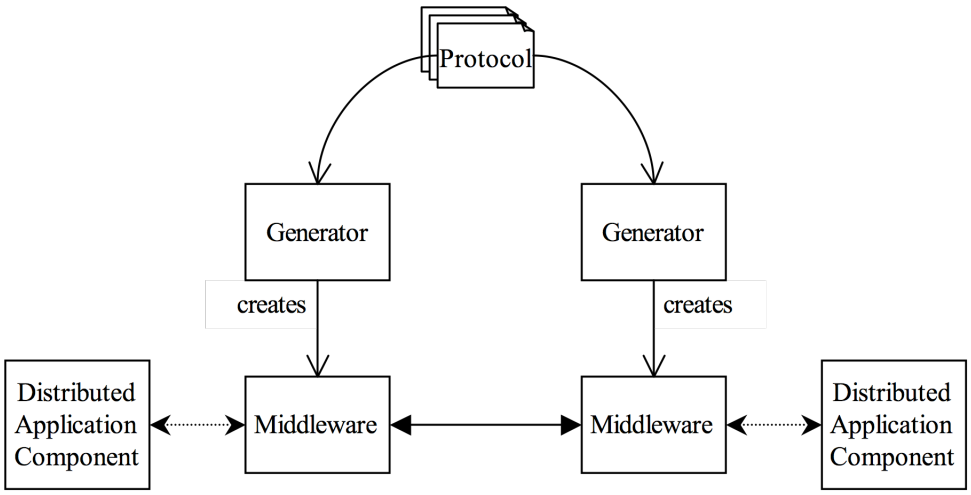

Figure 10: Middleware generation from protocol specification

Of course, this now requires each generator to be aware of the middleware component generated by the other, even before the first message of the protocol is sent, since the middleware components are capable of communicating with only a specific version of the common protocol (by design). Some coordination is then necessary in order to generate the correct middleware. One way in which to accomplish this coordination is to negotiate the version of the protocol to use beforehand. In Figure 11, negotiator components have been added for just such a purpose. The specification is passed from one side to the other during negotiation if necessary.

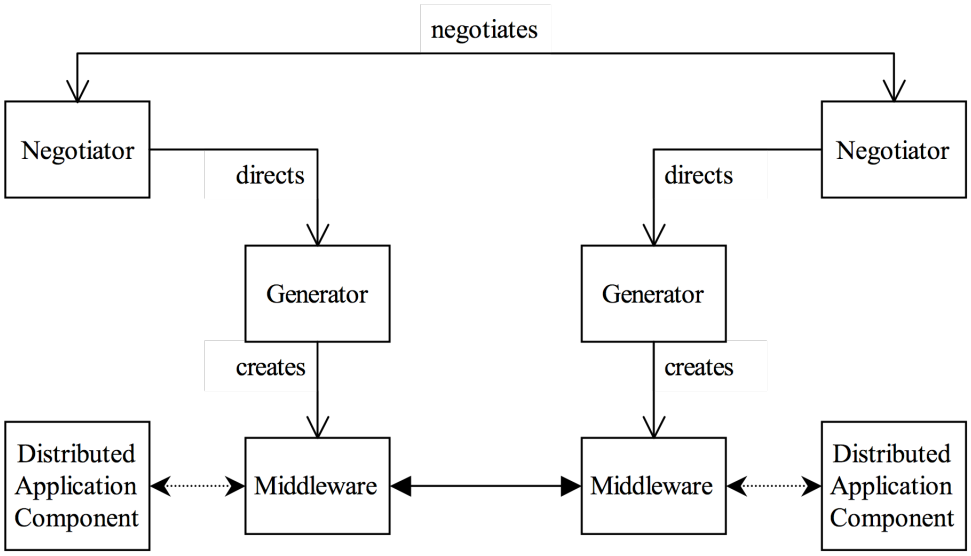

Figure 11: Protocol negotiation

This completes the framework that supports communication between two components, either one of which may have been upgraded to an evolution of the protocol in question. Notice that the diagram is beginning to reflect two mirrored images of Figure 4.

Irrespective of how they are instantiated, the middleware components must communicate with each other using the determined version of the protocol, and they must further interface with the application components via the conceptual information (as events) carried by the protocol. The middleware components are thus translators of a protocol into its conceptual units and *vice versa*; they are required to both parse protocol messages into semantic concepts as defined by the protocol and compose such concepts into legal protocol messages. Automatically generating components with this capability is the focus of this work. Figure 12 replaces the generic

middleware components with our notion of a translator as the pairing of a parser and a composer.

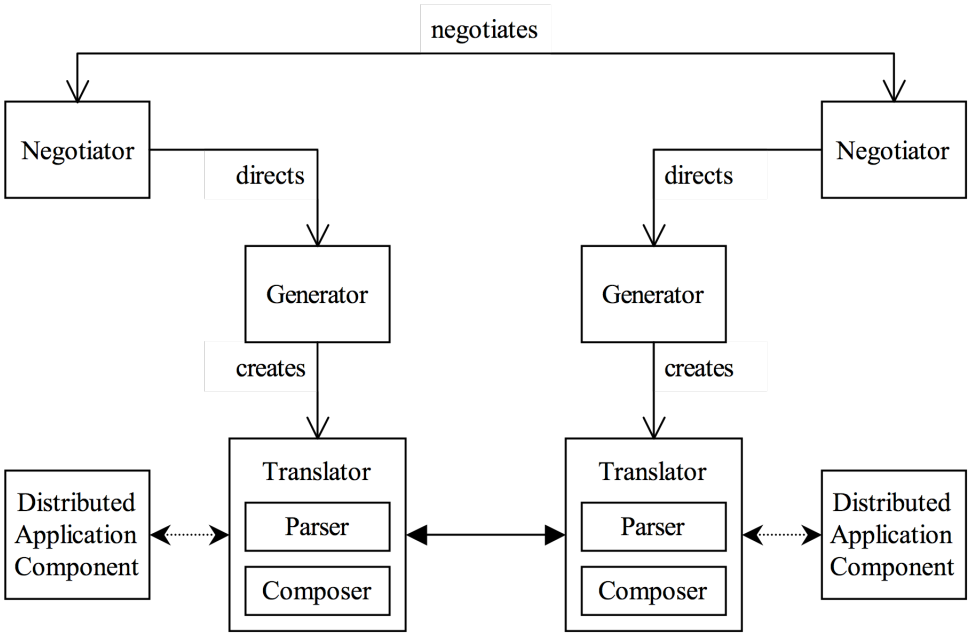

Figure 12: Translator parsing and composing

The messages defined by a protocol are generally not passed among the distributed components without regard to other messages, however; the protocol is used for communication of discrete and deterministic units that we refer to as interactions, which define sequences of messages sent and received among the components. Interactions are thus stateful, and the relevant state is altered in specific ways according to the ordering of messages, both incoming and outgoing. We thus require translators to track this information as well. Figure 13 adds this aspect of protocol translation.

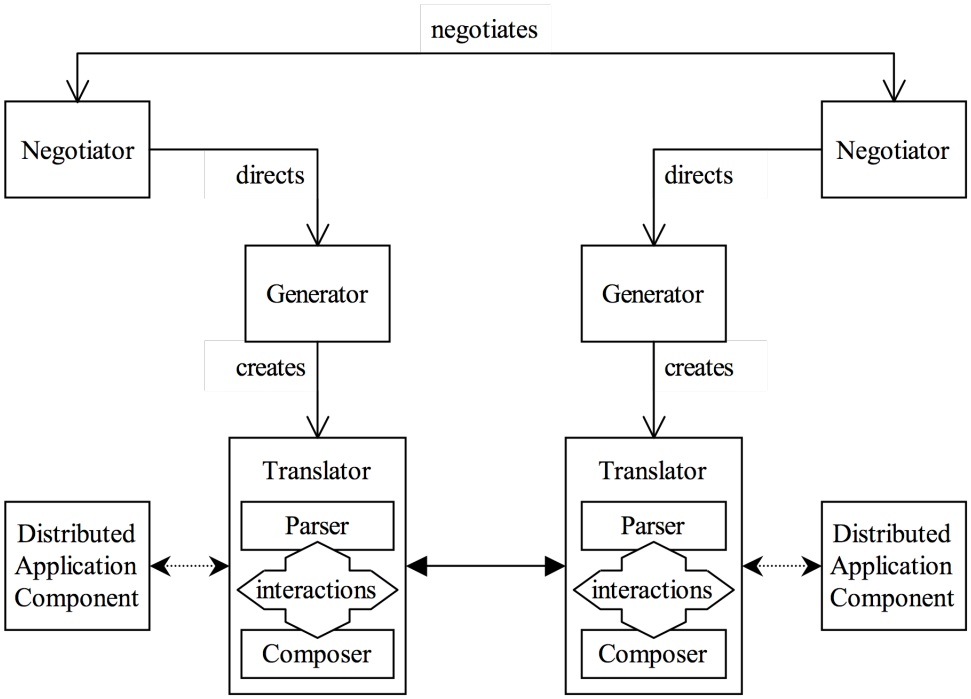

Figure 13: Translator interaction state tracking

Although not shown in the diagram, the transition of state during an interaction is also conceptual information of the protocol, as is a concept with which a token might be associated; consequently, this information should also be provided to the corresponding application component.

Like a translator, each distributed application component connected to a translator is also a producer and consumer of events that encapsulate semantic protocol concepts. Unlike a translator, which produces events via a parser, an application component produces events somewhat haphazardly, given that the component is (from the perspective of the translation system) the ultimate originator of such information, and its processes are not only unknown to the translator, but neither are they necessarily solely determined by the protocol, especially the version

of the protocol known to the translator. The consumption of events by an application component, however, more closely parallels the function of a translator's composer, and so we assign a conceptual architectural element, an event handler, to manage this aspect on behalf of the application component, shown explicit by Figure 14.

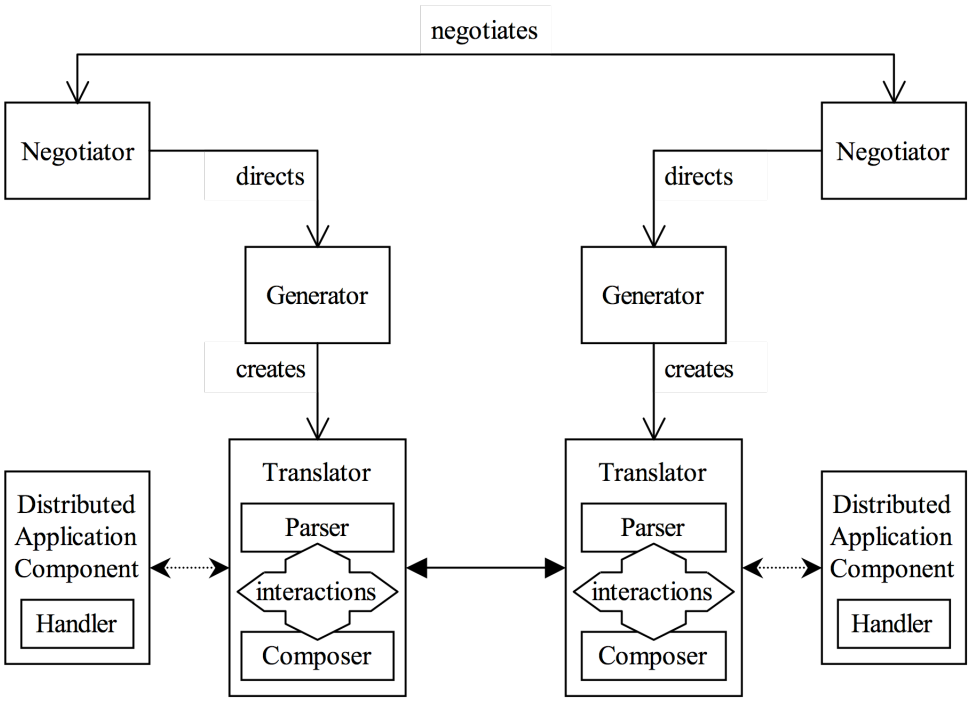

Figure 14: Application component handler

We have now derived all of the major components of Figure 4 (in mirrored duplicate), glossing over the details of input/output from the translator and the breakdown of protocol specification. The remainder of Figure 4 deals with the coordination among these components. The coordination is different depending on whether the translator (as proxy for the application component) initiates the connection over which the interaction takes place. (An interaction may take place over multiple connections, even multiple connections between the same two

components, but for the sake of introduction we will discuss only the simplest case.) Of the two sides of the figures, conceptually separated by a machine boundary, the side (by convention, the left side) that initiates the connection will be referred to as the *client*, whereas the side (by convention, the right side) that listens for the connection will be referred to as the *server*. Note that it is entirely possible for a single application to be both a client and a server, even within the same interaction, albeit from different perspectives.

3.1. Client-Side Coordination

Coordination of the components of the event-based translation system from the client-side perspective begins with the client application intending to establish a connection in order to initiate an interaction with the server. (We will, for the sake of simplicity, assume that the client will be sending the first message of the interaction, though this need not be the case, e.g., [KR01].) The client first suggests a protocol (or protocols) to the negotiator, requesting a pair of relevant addresses; one address is the "application side" of a client-side translator, and the other address is the "protocol side" of a server-side translator; note that the latter address is specific to a role, representative of the server application with which the client application desires to interact.

The negotiator must then provide the pair of addresses. If a client-side translator usable by the application (according to the suggested protocol) has already been generated and is available, the negotiator can simply use the corresponding address as the first of the two requested by the client application; otherwise, the negotiator must direct the client-side generator to create an appropriate translator

based on a protocol specification that is either known, appropriated from the client application, or otherwise obtained[10]. The second of the two addresses, that of the "protocol side" of the remote translator, might also be readily available (perhaps in a handy cache of such addresses); otherwise the client-side negotiator must contact its server-side counterpart in order to request the address. Like the protocol specification, the address of the server-side negotiator could be known, provided by the client application, or otherwise obtained (by means of a naming service, for example).

Once the client application is provided with the two addresses, it can connect to the local translator using the first address, provide the second address to the translator, and request a session that will last the duration of the interaction. The application then provides concepts (abstractly as events, recall) to the connected translator for composition into the initial message. After the application is finished providing concepts, a connection is opened from the client-side translator to the server-side translator, and the initial message is composed and sent.

This sequence of operations—from the client application contacting the negotiator to the client application connecting to the translator—is shown in Figure 15. The transmission of concepts from the application to the translator and the transmission of the message from one translator to the other, both of which occur

---

[10] The problem of resource discovery, characterized and analyzed for decades (e.g., [BDS93]), will not be addressed by this work in any significant detail, particularly given that the specification of any protocol used for communication must be known to at least one entity involved, although perhaps implicitly. In order to ensure that our resource discovery problem is trivial, we merely need to require that any application initiating an interaction be able to provide the specification for any protocol that it suggests; the specification can then be propagated via the negotiation system to all relevant entities.

after the coordination among the client-side components has been completed, are shown independently.

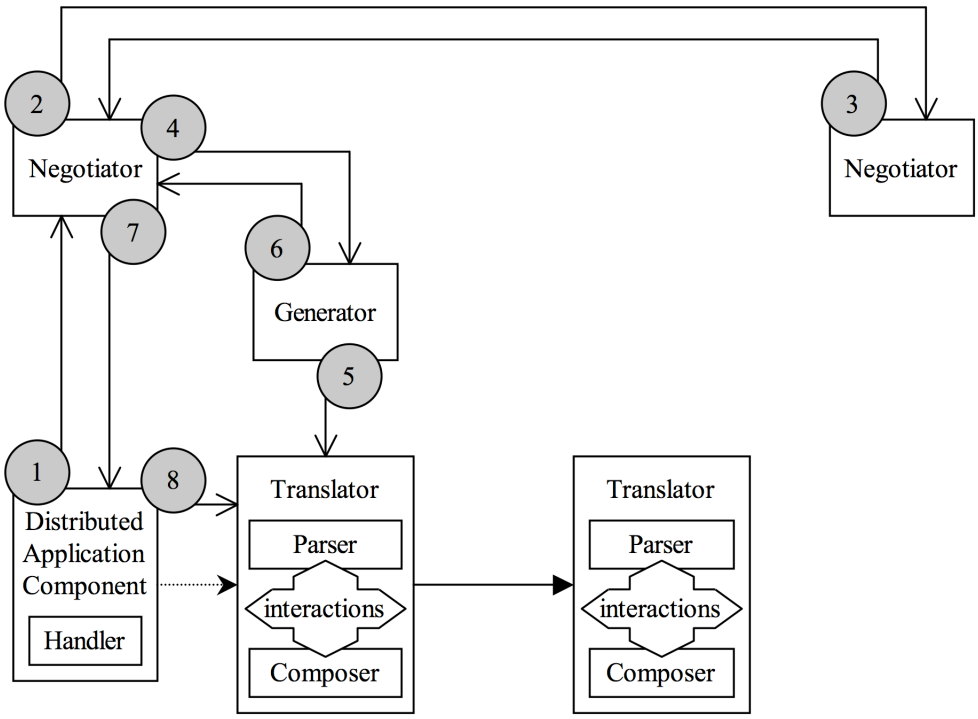

Figure 15: Client-side coordination

Summarized, the steps outlined by the diagram are as follows:

1) The client application sends the address of the server-side negotiator, the role implemented by the server application with which it wishes to interact, and a suggested protocol (name and version) to the local negotiator, implicitly requesting a pair of addresses.
2) The client-side negotiator contacts the server-side negotiator, requesting the role-specific address of a server-side translator specific to a protocol that the negotiators will determine is the most appropriate. Arbitrary communication may take place during the negotiation.

3) The server-side negotiator returns the relevant address to the client-side translator.
4) The client-side negotiator directs the generator to create a client-side translator specific to the determined protocol.
5) The generator instantiates and starts an appropriate translator.
6) The generator returns the "application-side" address of the new translator to the client-side negotiator.
7) The client-side negotiator returns a pair of addresses to the client application, the first being the "application-side" address of the client-side translator and the second being the relevant, role-specific address of the server-side translator.
8) The client application sends the role-specific address of the server-side translator to the client-side translator, implicitly initiating a session for a new interaction.

This coordination must be performed once per interaction; note, however, that the generation of the translator need only be performed once, and further optimizations (such as caching) could all but eliminate this relatively complex process altogether. Of course, this is only conceptually how the coordination among the client-side components is accomplished in the general case. The actual process as performed by an implementing system may simplify certain steps (e.g., the assumption of a well-known translator port) or perform intermediate operations (e.g., the retrieval of a protocol specification from a third party), as long as the conceptual coordination occurs as designed.

3.2. Server-Side Coordination

Although the server (right) side of Figure 14 is a mirror image of the client (left) side, the coordination required is much different. Rather than initiating interactions, the server is (conceptually) waiting for them to be initiated; it must therefore announce its intention to satisfy certain roles (if it is willing to satisfy the roles in a server context) to a corresponding negotiator, prior to participating in any interactions.

Once the server application has registered itself with the negotiator, the negotiator merely waits to be contacted (by another, remote negotiator) with a request for a component that implements the role. (This is, of course, a form of service-oriented computing, made popular by the relatively recent surge in Web services, though the protocols usable by a Web service component are considered static [BHM+03] [CKM+03].) Once the request has been made, the server-side negotiator determines if an application has registered itself for the role in question, and a negotiation for the appropriate protocol takes place.

If a relevant server-side translator is available, the role-specific address can be returned immediately. Otherwise, the negotiator must direct the generator to create a new translator specific to the negotiated protocol. Once this has been done, the generator can return the appropriate address to the server-side negotiator, which in turn returns it to the client-side negotiator.

The translator is then waiting for a connection to be established and for a protocol message to be sent by the client-side translator. Immediately after the connection is made, before the message is received, the server-side translator must

request the address of a registered application that implements the role for which the message is intended. If no such application exists, the message can be refused. Otherwise, the translator connects to (the handler of) the server application, informing it of the role that it will be assuming. The message can then be read and parsed, the events routed to the application.

This sequence of operations—from the server application registering itself with the negotiator to the server-side translator opening a session with the application—is shown in Figure 16. The transmission of the message from the client-side translator to the server-side translator and the transmission of protocol concepts to the application, both of which occur after the coordination among the server-side components has been completed, are shown independently.

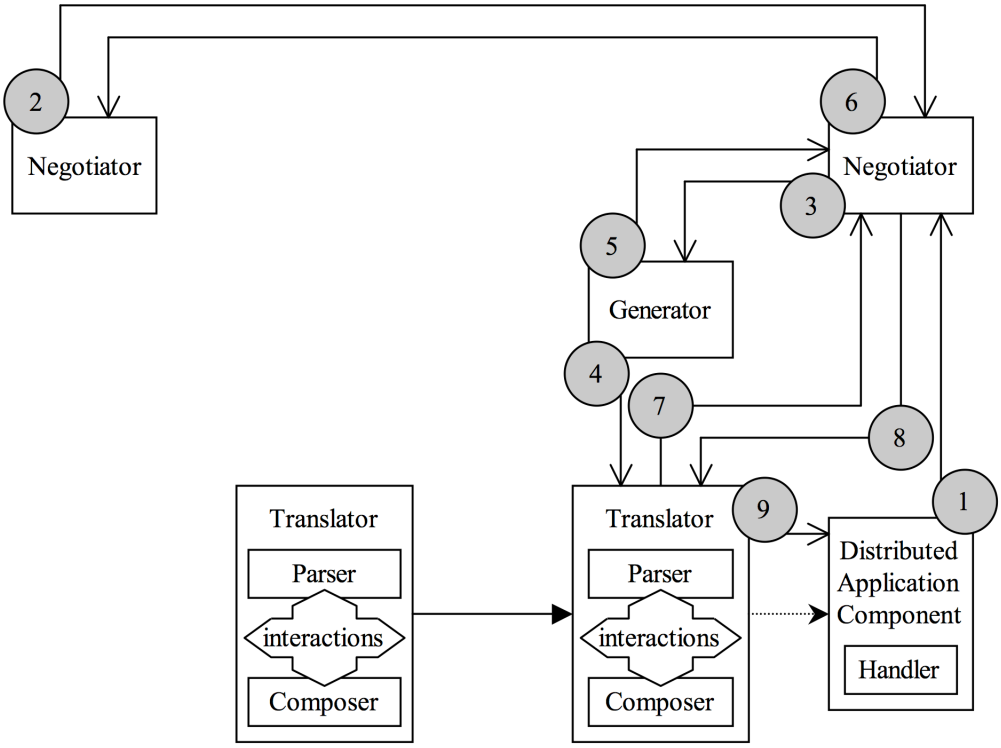

Figure 16: Server-side coordination

Summarized, the steps outlined by the diagram are as follows:

1) The server application sends to the server-side negotiator a role (or set of roles) of a protocol (or protocols) that the application is willing/able to satisfy, in addition to an address at which a translator seeking to use the application as a particular role can connect to the handler of the application. The server application and the server-side negotiator are now waiting for a connection to be initiated by a translator or another negotiator, respectively.
2) A client-side negotiator connects to the server-side negotiator, requesting a role-specific address for a server-side translator. The negotiators will

determine the most appropriate protocol to use for the coming interaction. Note that this step is the same as step #2 for the client-side coordination.

3) The server-side negotiator directs the generator to create a translator specific to the determined protocol.
4) The generator instantiates and starts an appropriate translator.
5) The generator returns a list of the role-specific addresses—one for each role that might wait to be contacted as part of an interaction—of the new translator to the server-side negotiator.
6) The server-side negotiator returns the relevant address to the client-side translator. Note that this step is the same as step #3 for the client-side coordination.
7) The client-side translator establishes a connection with the server-side translator, which does not yet read any message being transmitted over the connection. Instead, the server-side translator requests from its managing negotiator the address of the application implementing the appropriate role, as determined by the address at which the connection from the client-side translator was made.
8) The server-side negotiator returns the address of the server application implementing the indicated role.
9) The server-side translator connects to the server application, initiating a session for the interaction.

At this point, the message can be read from the connection and parsed, with concepts being sent (as events) to the handler of the server application. As with

client-side coordination, this coordination must be performed once per interaction, though the translator generation need only be performed once and further optimizations can be employed.

3.3. Alternate Configurations

As described, it would appear that our technique requires an application component to cooperate directly with the event-based translation system. In fact, that is just one way to make use of our approach. We are also able to apply the technique without requiring modification to pre-existing applications. In such cases, the components will communicate using the full protocol, and the role of the event-based translation system will be reversed.

As already implied, an event-based translator can be thought of architecturally as a bi-directional pipe, with messages of the protocol coming in and going out one side (the "protocol side"), and protocol concepts (as events) coming in and going out the other side (the "application side"), as shown in Figure 17.

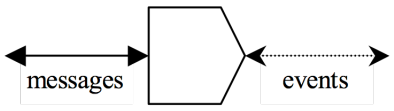

Figure 17: Translator as a bi-directional pipe

As is our convention, the solid line indicates message communication, and the dotted line indicates event communication. Also recall that a translator is specific to a single version of a single protocol.

Such a translator can then be connected to an application component in two different ways, depending on whether the component is "aware" of the event-based

translation system or not. Given any two distributed components that are acting as the end points of a communication, there are then four distinct configurations possible, effectively falling into three categories: both communicating components are aware, only one component is aware, or neither component is aware. We depict the categories in Figure 18.

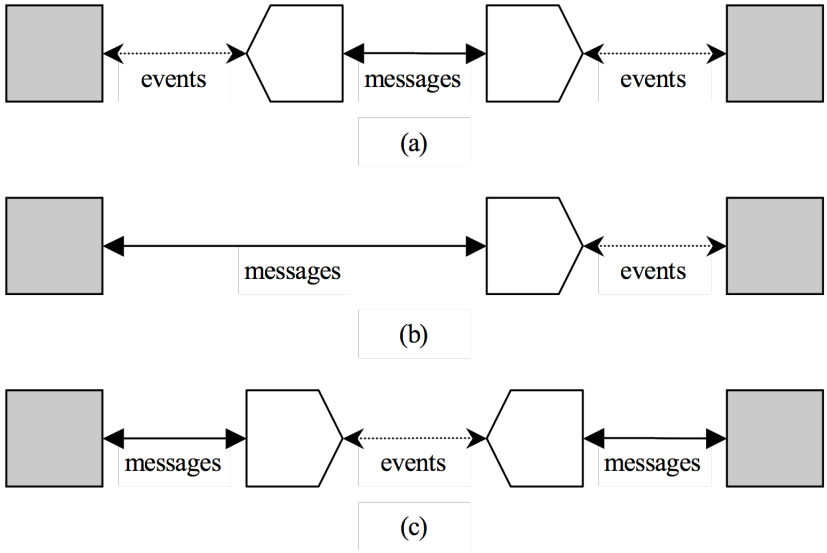

Figure 18: Alternate configurations

The first case, (a), is the ideal situation, since neither the protocol nor the protocol version being used is fixed by either distributed component. All message parsing, message composition, and interaction-state tracking is performed by the translators, so each component can be dynamically connected to different translators, and each message of the protocol is composed only once and parsed only once. Again, note that the translators must both be specific to the same protocol. The respective negotiators (not shown) will determine the protocol to use and direct the

corresponding generators (also not shown) to dynamically create the translators if necessary.

In the second case, (b), the left-hand component is dependent on the syntax of the protocol, but the right-hand component is not, and can therefore be dynamically connected to different translators. Again, each message of the protocol (communicated in either direction) is composed only once and parsed only once. The translator must be specific to the same protocol on which the left-hand component depends; no negotiation takes place, and the left-hand component is connected directly to the translator as if it was connected to the right-hand component. (An analogous situation occurs when the left-hand component is aware and the right-hand component is not, and thus we have not bothered to depict this case.)

The third case, (c), deals with the situation in which neither component is aware of the event-based translation system. In this case, the aware "component" to which each translator is connected is, in fact, another translator (possibly with an intervening filter for concepts passed between the translators). The protocol used by each component is fixed; the advantage this situation has over direct communication between the components is that the protocols used by each component can be different, yet the components can still communicate. The disadvantage is that, since potentially two different protocols are being used, each communication requires composition and parsing in both the protocol used by the right-hand component and the protocol used by the left-hand component, effectively doubling the amount of translation work required. In practice, both translators would exist on one side of the

abstract machine boundary, so that event communication is local. Again, no negotiation takes place.

In Chapter 7 we will examine two case studies, one of which embodies the first scenario (HTTP) and the other of which embodies the third scenario (SIENA).

3.4. Translator Generation

In the first scenario of Figure 18, both client-side coordination and server-side coordination require the run-time generation of a translator that is specific to the protocol used for communication across the logical machine boundary. In the last two of the three scenarios, translators must instead be generated as part of the configuration of the distributed application, prior to the activation of (or at least the communication of) the application. Regardless, this generation requires the input of the relevant formal protocol specification, and produces as output the relevant elements for event-based translation, incorporating them within the (possibly conceptual) composite component of a corresponding translator. In the dynamic case, the new translator is then started as an active component of the event-based translation system; in the preconfigured cases, the new translator is started as part of the overall distributed application. The function of the generator is summarized by Figure 19.

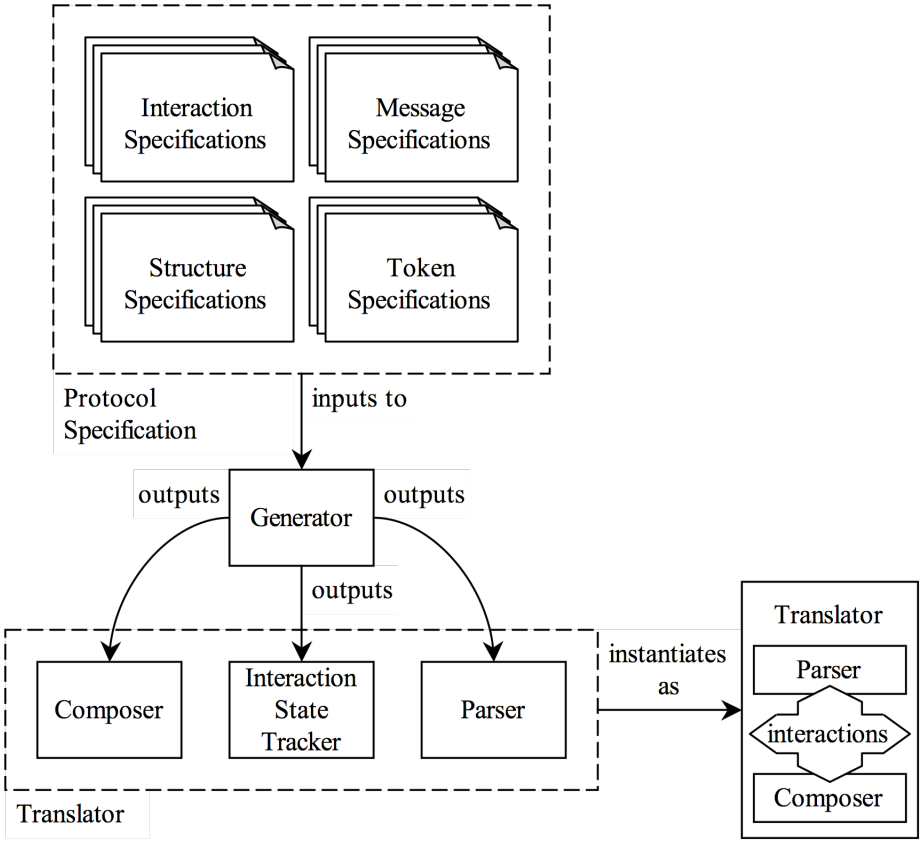

Figure 19: Generator responsibility

How the elements are implemented or even arranged in practice can vary, as long as the translator obeys the general contract as outlined by this work. In Chapter 6, we will explain in detail how various artifacts are used by the prototype in order to implement this process.

4. Protocol Specification

4.1. Principle

The automatic generation of a protocol recognizer and the adaptability of applications to deal with an evolving protocol rely on a formal specification of the protocol. Aside from the formality, the primary requirements of such a specification are three-fold:

1) Allow a protocol to be specified (i.e., provide a formalism that covers both the parsing and the composing of protocol messages, in addition to the larger context of message communication sequences).
2) Allow a protocol to be *conveniently* specified.
3) Isolate the effect of changes to a protocol from one version to another.

Each of these aspects will be discussed in turn.

4.1.1. Specification

The primary requirement of any formalism, in order for the formalism to be at all useful, is that it must be capable of specifying what needs to be specified. For the purposes of dynamic protocol evolution, the requirements for specification, roughly, are:

1) Specify interactions such that a component capable of *identifying and tracking* such interactions can be automatically generated.
2) Specify messages such that a component capable of *deconstructing* such messages can be automatically generated.
3) Specify messages such that a component capable of *constructing* such messages can be automatically generated.

Of course, program code is itself a formalism capable of fulfilling the above requirements. Clearly, then, these requirements are not sufficient for a useful system capable of dynamic protocol evolution, though they are necessary. Instead, we require a formalism that not only satisfies the task but facilitates it as well.

4.1.2. Convenience of Specification

The convenience with which a protocol is specified using a given specification language or other formalism (such as code) is subjective, and therefore difficult to submit to any rigorous analysis. However, one can still make the broad assertion that, for a formalism to be relatively useful for a task (besides fulfilling the task), it must be somehow more convenient than other formalisms used for the same task.

The task in question is dynamic protocol evolution, for which we require the specification of a protocol (that satisfies certain criteria with respect to features of dealing with the protocol). While other protocol specification languages clearly exist (e.g., CSP [Hoa85]), these languages either do not fulfill the task (specifically regarding the specification of messages with respect to composition) or they are capable of fulfilling it only in a roundabout way. Eiffel, for example, is a full-fledged

programming language [TC01] that, while sufficient to formally specify both the interactions of a protocol and (programmatically) the messages passed, does not provide a convenient notation for such.

Instead, a formalism that provides the necessary features (in an intuitive notation), and little else, is preferred. To this end, a specification language (or in this case, as we shall see, a suite of specification languages) particular to this task can be developed. This ensures that the formalism is not only capable of expressing those features of protocol specification required for the task, but also that the formalism can do so without being burdened by any additional complexity introduced by the presence of unnecessary features.

4.1.3. Isolation of Changes

Now, supposing the existence of a set of convenient formalisms that satisfy the given requirements of protocol specification, we impose an additional criterion to filter them further: A formalism for protocol specification must be capable of isolating changes to the specification of a protocol from one version of the protocol to another.

For an incremental change, it is undesirable that the corresponding specification be altered fundamentally, unless the change is also fundamental. In other words, we prefer that the scope of a change within a protocol specification be reflective of the change to the protocol.

For this reason we view a protocol specification as many parts, in order to define “boundaries” of change and prevent small changes from propagating into large ones. Figure 20 reproduces Figure 2, showing the abstract four-part specification of a

protocol's syntax and the reference dependencies among the various parts. Note that tokens can depend on other tokens, and structures can depend on other structures (or, even, recursively on the same structure, as explained below).

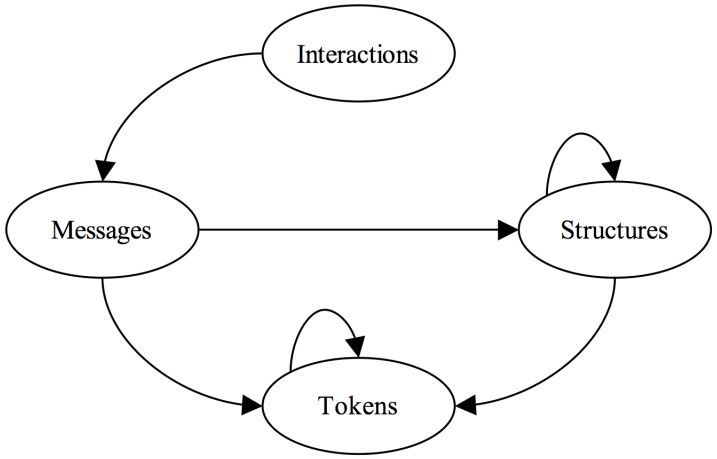

Figure 20: The four parts of protocol syntax specification

As implied by the use of the word "token", protocol messages (and structures and tokens) are specified using the same "document" specification techniques pioneered in the 1960s (e.g., [Knu62]). Interactions, instead, require a specification that more conveniently expresses coordination.

Theoretically, we have a number of choices for each level of expressiveness for the specification language for each of these parts. It is unlikely, however, that a protocol message will require the expressiveness of, say, a programming language, or that a protocol interaction will require more than a handful of communications. Consequently, we will allow (in most cases) the expressiveness to be restricted to something significantly less than a fully context-free grammar, in both theory and practice.

4.1.3.1. Interactions

An interaction describes the sequence in which messages must be passed in order to satisfy a proper "use" of the protocol. Since each interaction is expected to involve relatively few communications, we limit interactions to those that can be described by a deterministic finite state machine. Consequently, interactions are considered to be describable by a simple state machine language whose expressiveness is regular.

Note that an interaction can occur over more than one connection, even between the same two components (e.g., FTP [PR85]). This has the potential to introduce nondeterminism into an interaction. (See Section 5.1.2 for more discussion.)

4.1.3.2. Messages

Messages are compositions of tokens and structures. While a message may have an overall definition that is no more complex than that of a structure (see below), a message represents the highest-level structure and must incorporate features to handle at least one special case of context sensitivity, discussed shortly.

4.1.3.3. Structures

Like messages, structures are compositions of tokens and other structures; both messages and structures therefore carry relational information in contrast to the basic information of tokens. Structures are considered to be describable by a language that is something less than a fully context-free language, even less than LR(1). What expressiveness is chosen will depend on the availability and suitability of tools that will parse such a language.

4.1.3.4. Tokens

Tokens are at the lowest level of a protocol specification and, as expected, capture the basic informational "pieces" of each message. A token is a meaningful element of the syntax insofar as smaller units (unless they are themselves tokens) carry no information outside their context. Tokens are considered to be describable by a regular language.

4.2. Interaction Considerations

The specification of an interaction amounts to a DFA (deterministic finite automata), with each sent and received message defining a transition between two states of the DFA. Each role has its own (possibly partial) perspective of the DFA, however, and consequently each state of the DFA may be simultaneously labeled with several different states, possibly one for each role.

For example, the following defines a complete interaction using the interaction grammar specification language (see Appendix B) created for this work:

```
[messages]
  M1 M2 M3
[roles]
  example 1.0 roles   // defines A, B, C
[states]
  example 1.0 states  // defines A0-A3 B0-B2, C0-C1
[initializations]
  A A0
  B B0
  C C0
[communications]
  M1: {A0} A A1 -> {B0} B B1
  M2: {B1} B B2 -> {A1} A A2
  M3: {A2} A A3 -> {C0} C C1
[end]
```

This interaction is represented pictorially by Figure 21. Each circle of the diagram represents a role involved in the interaction, and each arc represents a

message being passed from one role to another; messages are sent (and received) in the natural order.

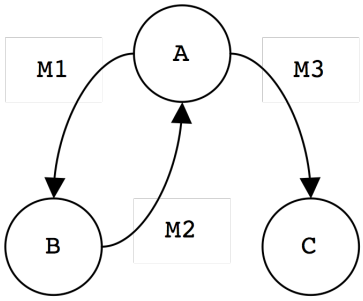

Figure 21: Example interaction

The DFA of the interaction is given in Figure 22, along with the perspective of the interaction from each of the roles `A`, `B`, and `C`.

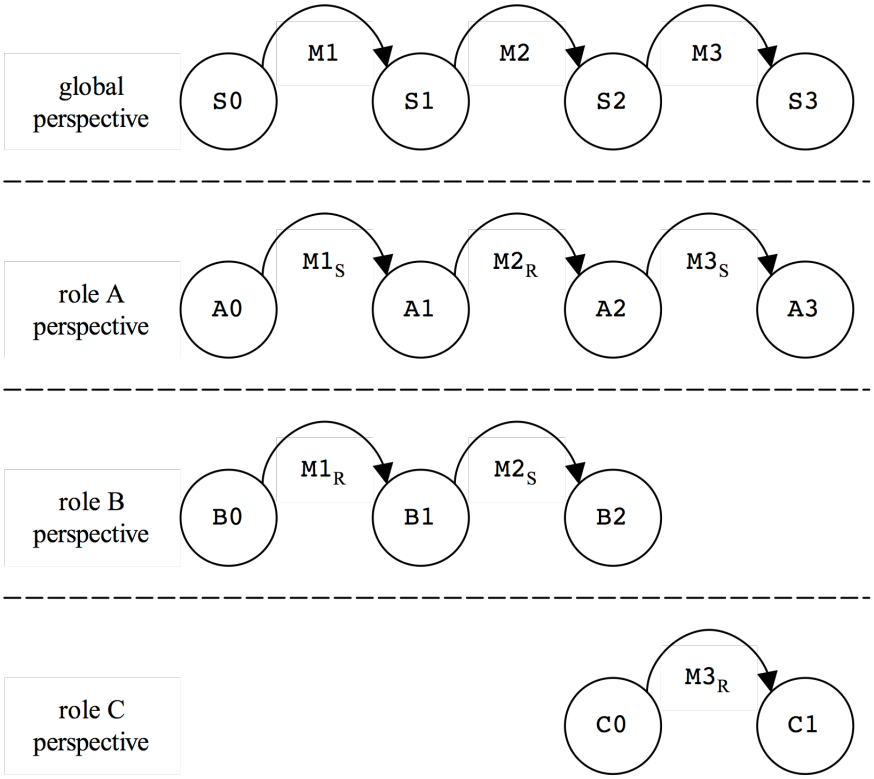

Figure 22: Example interaction DFA from several perspectives

Each circle of the diagram represents a state (corresponding states of the various perspectives are vertically aligned), and each arc represents a transition from one interaction state to another. The states of the global perspective are the typical states of the DFA, whereas the states of the role perspectives are the states of the interaction (as perceived by the relevant role). The messages labeling the transitions in the role perspectives are subscripted to indicate a message being sent (S) or a message being received (R), since state transitions (from the perspective of an individual role) are dependent on such information. Note that the perspective of the interaction from both of the roles `B` and `C` is partial.

An interaction specification thus defines several perspectives of the same interaction, one for each role involved in the interaction. The messages passed among the roles are identical (though the function of sending/receiving is reversed) in overlapping portions of the role perspectives, but the names of the states are relative with respect to each perspective. Finally, note that the global perspective (and its states labeled `S0`, `S1`, `S2`, and `S3`) is not explicitly defined by the interaction specification; however, it can be derived as the "sum" of the individual role perspectives.

4.3. Parsing Considerations

To summarize the expressiveness of protocol syntax, as described in Section 4.1: Interaction syntax is restricted such that it can be described by a state machine (regular) language; message and structure syntax is restricted such that they can be described by something less than a fully context-free language, though message syntax (as a whole) requires an extension to deal with a special case of context sensitivity; token syntax, on the other hand, is restricted such that it can be described by a regular language.

With regard to the expressiveness of messages and structures, we will be more specific by placing the restriction of LL(1) expressiveness on them (the extension of context sensitivity for messages notwithstanding). As previously mentioned, it is unlikely that protocol messages will require the expressiveness of a programming language, which are typically describable by an LALR(1) grammar, perhaps with some minor extension. However, placing a limitation of non-ambiguity seems too strict, so an LR(0) (or even an SLR) grammar also seems unsuitable. Therefore, we

hypothesize that an LL(1) language will be descriptive enough for the vast majority of cases, and so make this restriction for the purpose of explicitness.

Of course, in practice, any specification language that is *more* powerful is also suitable; we place these upper bounds of expressiveness on the (expected) syntax of a protocol, not on the grammar of the specification languages that will describe the syntax. (Restrictions on the syntax of the specification can be verified when the specification is evaluated, and need not be enforced explicitly by the language itself.)

Thus, we have defined the limits of expressiveness necessary to determine what classic parsing techniques can be used with respect to the decomposition of protocol messages. Some further conditions warrant attention, as well.

4.3.1. Conveyance of Data as Concepts

An important consideration for parsing is the notion that an application should be able to deal with the conceptual information captured by the syntax of the protocol, rather than with the actual syntactic details of the protocol. To this end we view "important" elements of a protocol to be associated with (known) concepts. Not only does this allow an application using the protocol to ignore elements of the syntax that are intended purely as an aid to the syntax itself (such as, for example, separators, or any other "syntactic sugar"), but the application then need only concern itself with the presence/absence of such information, regardless of how such information is captured by the syntax.

However, this requires a mechanism by which elements of the syntax can be properly associated with conceptual information. This mechanism must be built into the specification languages used to describe the protocol.

4.3.2. Termination and Context Sensitivity

A particular kind of context sensitivity dealing with counted termination can occur within protocol messages (among other structures). Thus, the specification language for messages (and structures and tokens, as well, given that the level at which the context sensitivity is considered can occur anywhere within the message) must provide a mechanism to expand outside its context-free range, in order to accommodate this context-sensitive feature of message syntax.

Without regard to ambiguity/non-ambiguity, there are four methods for determining when a composite structure (token, structure, or message) terminates:

1) The structure is terminated by a pre-determined symbol (for example, the closing tag of an XML document [BPS+04]).
2) The structure is terminated after a pre-determined number of repetitions (sometimes a special case of the previous method).
3) The structure is terminated after a number of repetitions specified by an earlier structure (for example, the number of scanlines in the tag image file format of an Internet fax is determined by the value of the "ImageLength" field [BVM+04]).
4) The structure is terminated after a symbol defined by an earlier structure (for example, the separator of a Multipart-MIME content [Lev97]).

The first two of these are context free and can be thus captured by context-free languages, whereas the last two are context sensitive. (Note that the last two are technically context free in the strict definition of the Chomsky Hierarchy, though context sensitive in practice). Given the restrictions of each part of our protocol

syntax, the first two are allowable (provided the construct falls within the restriction). The last two are disallowed in practice, so the languages chosen to specify protocol syntax must provide a special mechanism in order to deal with these, if they are to be considered.

While it would be desirable to enable both of these latter two methods, detecting a context-sensitive terminator is more difficult than counting, so for the time being we dismiss the last case and require only the first three to be provided by the specification languages.

Of course, even context-sensitive counting is not without issues, the least of these being the format of the count as specified earlier in the message. We leave the possible formats and their corresponding limits as an implementation option.

4.4. Composition Considerations

The inverse of parsing is composition. If the application is to be ignorant of the specifics of protocol syntax, not only must the parsing activity be relegated to an automatically generated component, but so too must the composition activity.

The expressiveness of the expected protocol syntax was determined in the above section on parsing considerations, so that appropriate parsing techniques could be identified. While the same attention has not historically been given to the complementary technique of composition, the composition of protocol messages nevertheless must permit the full range of expressiveness. However, while the specification languages must provide the means by which both the parsing considerations and the composition considerations can be satisfied, since parsing and composition work in tandem, only the generation of one of the components (either the

parser or the composer) need enforce the boundaries of expressivity. This implies that we can use the understood issues of parser generation to enforce restrictions on expressivity, while ignoring the same for composer generation (since only one must enforce the restrictions), so long as we guarantee that the general form of a composer will accommodate languages that are at least as powerful as those desired.

For example, suppose we restrict a syntax to LL(1), but the specification language we are using allows the definition of languages that are more powerful than LL(1). The generation of a parser for the language, based on the specification, can validate this restriction; in so doing, the generation of the composer could ignore the issue altogether—provided the resulting composer can always handle languages that are LL(1) or greater in expressiveness—since the well understood process of parser generation is able to independently perform the validation. (Likewise, the process of composer generation could independently enforce this restriction, but the process of parser generation seems the more logical choice to detect such specification anomalies, given its historic understanding.)

4.4.1. Conveyance of Data as Concepts

Just as the specification languages must allow the association of syntactic elements with concepts for the purposes of relaying concepts to an application as a message is parsed, so too must these associations be made for the purposes of allowing an application to relay concepts so that a message might be composed.

Note that this requires the precise syntax of a token with respect to the concept with which it is associated to be somewhat symbiotic. This will be explained later (*q.v.*, Section 5.3.5) in greater detail.

4.4.2. Default Values

Given that an application does not need to be concerned with the full spectrum of syntactic details of protocol messages to be sent (whether by ignorance or reticence), it may be that an insufficient amount of data (in the form of concepts, annotated with literal values) has been provided by the application in order to compose a valid protocol message. (For example, if a newer version of the protocol is being used, messages may require one or more concepts that the relevant application was not programmed to provide.) In this case, either no message can be composed or default literal values must be used for the necessary tokens for which no associated concepts (and associated literal values) have been provided.

The latter possibility is preferred. Consequently, the specification languages must include a mechanism by which default literal values for tokens can be specified, either explicitly or by calculation. There are several aspects of these defaults to be considered:

1) Should a default literal value be associated with a particular token at all?
2) Is the default literal value static? That is, is the default literal value (associated with a token) constant, regardless of any literal value provided with a matching concept by an application?
3) Is the presence of the token optional from the perspective of a completed message?
4) In the circumstance that no message can be composed, should the default literal value be “forcibly” included?

Each of these aspects of default literal values must be incorporated into the language that allows tokens to be specified.

4.5. Practice

As a necessity for formal protocol specification, we have developed a suite of languages to meet our requirements as given above. While other languages and tools (*viz.*, CSP [Hoa85], Eiffel [TC01], etc.) exist for formal protocol specification, these tools provide levels of detail that are extraneous from the perspective of the (relatively) simple functionality required, which makes interfacing with them for the purposes of a prototype system difficult. For example, while CSP is capable of describing interactions (including ours) on the most general level, its calculus allows for such features as nondeterminism (among a number of other capabilities provided by its various operators), which is disallowed for our simple interactions; Eiffel, on the other hand, is a complete programming language, making the extraction of relevant information problematic.

The suite of languages is divided roughly into two categories: those that deal with concept specification, and those that deal with grammar specification. See Appendix B for the technical specifications of each of these languages.

4.5.1. Concept Specification

Concepts, a major theme of this work, require their own specification, separate from the syntax of a protocol. Concepts encapsulate some *meaning* with which various aspects of a protocol's grammar are associated, and as such their specification is more esoteric than a formal grammar. However, given that protocols are expected to evolve yet the concepts are expected to remain (largely) the same

across evolutions, we would like to make concepts able to be referenced by any number of protocols. With this in mind, we allow each concept to be defined as a single, qualified name, with an associated semantic understood by the defining and referencing entities.

A concept specification language allows some class of concepts (to be associated with some element of a protocol's syntax) to be defined. Since several different aspects of a protocol's syntax must be related to concepts, we have consequently determined five orthogonal classes of concepts divided into two groups: interaction concepts (roles and states) and production concepts (messages, structures, and tokens). However, each of these languages is used merely to list a set of names, each of which (by itself) defines a concept; therefore, the specification of all five languages is identical.

4.5.1.1. Interaction Concepts

Interaction concepts are concepts used within the definition of an interaction; they include the roles that application components may assume during the interaction and the various states of the interaction.

Again, the corresponding language for both of these two concept classes is identical to the languages used for production concepts, merely allowing one to define a list of names.

4.5.1.2. Production Concepts

The production concepts are, as their name implies, concepts with which productions of the protocol grammar are associated. Explicitly, message productions

are associated with message concepts, structure productions are associated with structure concepts, and token productions are associated with token concepts.

The corresponding language for each of these three concept classes amounts to a single list, allowing for logical groupings of concepts within each of the classes. For example:

```
[concepts]
  METHOD_GET_NAME
  STATUS_200_NAME
  VERSION_MAJOR
  VERSION_MINOR
  PROTOCOL
  REASON_PHRASE
[end]
```

Each of the names between the tags above indicates a concept with which one or more productions (token productions, in this case) may be related.

4.5.2. Grammar Specification

Grammar specification allows some aspect of the syntax of a protocol to be defined. There are five grammar specification languages, one each for the overall protocol (i.e., the collection of interactions that defines the protocol as a whole), interactions, message productions, structure productions, and token productions.

Specification elements (be they interactions, productions, or concepts) are arranged into groups. Each specification element is named, and names are qualified by the (qualified) name of the group in which the element is defined. (Note that the grammar of the languages allows interactions, each type of production, and each type of concept to have a separate namespace.) However, elements are referenced only by unqualified names within the specifications, so an import/aliasing mechanism has been developed.

4.5.2.1. A Note on Aliasing

Before we discuss the particulars of the grammar specification languages, an introduction to the referencing and aliasing feature is in order. A specification written in one of the grammar specification languages can, and is most often required to, reference the elements of a specification written in another (or even the same) grammar specification language. These references are satisfied by what are referred to as "imports"; the elements of a group must be imported by a referencing specification if at least one element of the group is referenced. With the exception of interactions and messages (since interactions and messages are particular to a protocol specification, only one interaction/message is specified per group, and the name of the interaction/message is determined by the name of the group), an import can optionally be "aliased" and, as such, the elements of that import are referenced with an alias qualifier.

For example, suppose we are given the following message specification, using the language of Appendix B.1.3:

```
[tokens]
  example 1.0 tokens          // defines T1, T2
  example 2.0 tokens          // defines T1, T2
[structures]
  example 1.0 structures      // defines S
[concepts]
  example 1.0 concepts        // defines C
[productions]
  S (<T1> <T2>)
  M (S | S) = C
[end]
```

Without qualified names, the references to tokens `T1` and `T2` and both instances of the structures `S` are ambiguous. The specification has no way to disambiguate our intention to use version 1.0 of `T1` and version 2.0 of `T2`, for

example. Instead, the imported groups must be aliased as necessary (note that each possible context of alias also has its own namespace), and the references qualified as appropriate:

```
[tokens]
  example 1.0 tokens = V1     // defines T1, T2
  example 2.0 tokens = V2     // defines T1, T2
[structures]
  example 1.0 structures = S  // defines S
[concepts]
  example 1.0 concepts        // defines C
[productions]
  S (<T1@V1> <T2@V2>)
  M (S | S@S) = C
[end]
```

While the grammar of a specification language defines syntactic conditions and its use may prescribe some semantic conditions, this mechanism introduces several additional semantic conditions that must be enforced by an implementing system:

1) Each group should be imported only once, though it is not considered an error to import a group more than once with the same or different aliases.
2) The elements of each group are considered to have been imported by the group without being aliased; they can be referenced by other elements in the same group even before they are technically defined. The current group can even (but should not) import itself, though it is not considered an error to do so.
3) Each name referenced by a specification, whether or not it is qualified by an alias, must be associated with an element defined in an imported group of the appropriate type. This condition is critical; if violated, the protocol specification as a whole should be considered invalid.

4) Each alias used to qualify a name referenced by a specification must be used to alias the group in which the element associated with the name is defined. This condition is also critical; if violated, the protocol specification as a whole should be considered invalid.
5) The names associated with the elements defined in each group (in the same context) imported with alias *A* should be distinct; if not, only the last such element should be accessible (by qualification with *A*).

Note that other conditions can be detected and enforced (e.g., superfluously imported groups), though an implementing system should passively allow the violation of such.

The following subsections will discuss the five grammar specification languages, each of which allows for the reference to elements defined by one or more of the other nine specification languages.

4.5.2.2. Protocol Specification Language

The root protocol specification language is used simply to define a protocol specification. Since a protocol specification is ultimately defined in terms of its interactions (which reference messages, roles, and states, the messages of which reference structures, tokens, and message concepts, etc.), the protocol specification language is used merely to import the set of interactions that define a protocol. For example:

```
[interactions]
  GET
  POST
[end]
```

This example protocol specification imports the interactions `GET` and `POST` as the plenary definition of the protocol.

4.5.2.3. Interaction Grammar Specification Language

The interaction specification language is used to define the interactions of a protocol specification; each interaction specification written according to this language is an interaction grammar group. For each group, the messages, roles, and states used to define an interaction are first imported. Then, the state of the interaction is initialized (via a conceptual transition at the beginning of the interaction) for each role potentially participating in the interaction; this defines the "start" state of the interaction from the perspective of each role. Finally, each communication that can take place during the interaction is listed. As an example, the interaction specification of Section 4.2 is reproduced below:

```
[messages]
  M1 M2 M3
[roles]
  example 1.0 roles
[states]
  example 1.0 states
[initializations]
  A A0
  B B0
  C C0
[communications]
  M1: {A0} A A1 -> {B0} B B1
  M2: {B1} B B2 -> {A1} A A2
  M3: {A2} A A3 -> {C0} C C1
[end]
```

This specification first imports the messages `M1`, `M2`, and `M3` (each of which is defined as a separate specification), the roles of the interaction role concept specification `roles` (defined as part of `example`, version 1.0), and the states of the interaction state concept specification `states` (also defined as part of `example`, version 1.0). Each of the roles participating in the interaction is then assigned an

initialization state (i.e., the state of the interaction at the beginning of the interaction, from the perspective of the role).

Finally, the communications are defined, indicating messages sent among the various roles. The first communication, for example, states that message `M1` is sent from role `A` to role `B`. The state of the interaction from the perspective of role `A` must be `A0`, which transitions to `A1` once the message is sent; the state of the interaction from the perspective of role `B` must be `B0`, which transitions to `B1` once the message is received.

Note that messages, not message concepts, drive the state of an interaction; therefore, the messages imported by a group need not be associated with message concepts. Also, note that messages imported by a group cannot be aliased (in the same way that roles and states can be aliased); this is because each message must be specified in its own group, uniquely determined by the name of the message, and each message specification is particular to the same protocol to which the interaction is particular.

4.5.2.4. Message Grammar Specification Language

The message specification language is used to define the messages of a protocol specification; each message specification (and collection of supporting structure specifications) written according to this language is a message grammar group. The tokens, structures, and message concepts used to define a group are first imported. Then, the message is defined via one or more named productions. Only one production represents the message itself; all others are structure productions particular to the message (whether or not they are used in the definition of the message). The

name of any defined production (other than the message production) can be referenced (unaliased, without import) by other productions within the same group.

An example of a message specification was given in Section 4.5.2.1. Another (shorter) example, forming message `M1`, is shown below:

```
[structures]
  example 1.0 structures
[concepts]
  example 1.0 concepts
[productions]
  MS $ignore(<WS>) (S1 | S2)
  M1 (MS)+ = C
[end]
```

This specification does not import any tokens, but imports the structures defined in `structures` as part of `example`, version 1.0. The concept names defined in `concepts` are also imported. One message-specific structure, `MS`, is defined; the message-specific structure includes a directive to ignore the token `WS` as whitespace between the tokens that ultimately define the structure. Finally, the message `M1` is defined as one or more repetitions of `MS` and associated with the concept `C`; this association of a production with a semantic concept produces a kind of attribute grammar, though the attributes are not used as an aid to recognizing the grammar when parsing a string or to enforce any semantic constraint.

The productions defined in a message grammar group have the following features:

1) One production must have a name matching the name of the message grammar group. This production is the message production; all other productions are supporting structure productions.

2) The message production may not recursively reference itself, though supporting structure productions may contain recursive references. (If a production of the same name is referenced in the definition of the message production, the reference production will be assumed to be a structure production from the structure production namespace.)
3) Only the message production may be associated with a concept.
4) Only the message production may utilize the leading whitespace directives "regard" and "ignore". (Both the message production and supporting structure productions may utilize the same in-rule whitespace directives, however.) The "regard" directive indicates that whitespace is relevant to the tokens that define the message, whereas an "ignore" directive indicates that the given whitespace token should be ignored between tokens that define the message; more will be said regarding the implications of whitespace in Section 5.3.9.

Note also that this language allows the definitions of messages and structures that are (much) more powerful than LL(1). The condition of LL(1) must be enforced (with the exception of context-sensitive repetition/termination) during an analysis of the protocol specification prior to the generation of a translator for the protocol; if this condition of expressivity is violated, the protocol specification as a whole should be considered invalid.

4.5.2.5. Structure Grammar Specification Language

The structures specification language is used to define the structures of a protocol specification; each collection of structure specifications written according to

this language is a structure grammar group. The tokens, structures, and structure concepts used to define a group are first imported. Then, the structures are defined via named productions. The name of any defined production can be referenced (unaliased, without import) by other productions within the same group.

The definitions of a structure grammar group look much like the definitions of a message grammar group. An example specification for a structure grammar group is shown below:

```
[tokens]
  example 1.0 tokens
  example 1.0 more
[productions]
  S1 (<COUNT> $regard <SEP> $regard <VALUE>)
  S2 (LENGTH)
  S3 (S2 <CRLF>)*
[end]
```

This specification imports only tokens, from two different specifications; no other structures are imported, nor are any concepts imported (thus, no structures defined in this group will be associated with structure concepts). Three structures are then defined, using the imported tokens and structures defined within the specification. Notice that the structure S1 uses “regard” directives to indicate that any whitespace defined as an “ignore” token by a higher-level structure or message is relevant (i.e., whitespace must be matched as part of its tokens, and should not be ignored).

Production definitions of a structure grammar group have the following features:

1) Each structure production definition may recursively reference itself, provided the overall grammar remains LL(1).

2) Each structure production may utilize the in-line whitespace directives, “regard” and “ignore”.

Although a structure grammar group looks much like a message grammar group, there are few (though still significant) differences between the two languages:

1) Each definition of a structure grammar group is a structure production, whereas the definitions of a message grammar group are structure productions and a single message production that has the same name as the group.
2) Each definition of a structure grammar group may be associated with a structure concept, whereas the (structure production) definitions of a message grammar group may not be associated with a structure concept.
3) Each definition of a structure grammar group may reference itself (thus defining a recursive structure); while the supporting structure definitions of a message grammar group may do likewise, the definition of the message production may not.
4) Each definition of a structure grammar group may not include leading whitespace directives, whereas the message production definition of a message grammar group may include both a “regard” and an “ignore” leading whitespace directive.

Note that, as is the case with the message specification language, this language also allows the definitions of structures that are (much) more powerful than LL(1). Again, the limit of LL(1) expressivity must be enforced prior to translator generation.

4.5.2.6. Token Grammar Specification Language

The tokens specification language is used to define the tokens of a protocol specification; each collection of token specifications written according to this language is a token grammar group. The tokens and token concepts used to define a group are first imported. Then, the tokens are defined via named productions. The name of any defined production can be referenced (unaliased, without import) by other productions within the same group (though tokens may not be recursively defined).

For the sake of uniformity, the definition of a token grammar group looks much like the definitions of the other two grammar groups (messages and structures). An example of a token grammar group specification is shown below.

```
[concepts]
  example 1.0 types
[productions]
  SEP (":")
  COUNT $constant ("Count")
  VALUE $optional ("0" | ["1"-"9"](["0"-"9"])*) = COUNT
[end]
```

Token concepts are first imported from the token concept specification `types` of `example`, version 1.0. Three tokens are then defined, the last of them being associated with the concept `COUNT`. Note the "constant" composition directive for the token `COUNT` (not to be confused with the token *concept* `COUNT` or the literal value "`Count`"), which defines the value of the token as static, and the "optional" composition directive for the token `VALUE`, which indicates that the literal value associated with the token in an abstract composition tree (see Section 5.3) should not be initialized with any default literal value that might be calculated from the definition.

The token production definitions of a token grammar group have the following features:

1) Each token production definition may not recursively reference itself. Although recursive definitions do not necessarily violate the limit of regular expressivity, they are unnecessary; full regular expressiveness can be achieved through the use of the standard regular expression operators.
2) Each token production may utilize the leading composition directives "none", "forcible", "optional", and "constant" (though the combinations of such are limited by the language definition). A "none" directive indicates that no default literal value should be calculated for the token; a "forcible" directive indicates that the default literal value associated with the token should be used when a message is being forcibly composed (see Section 5.3.6); an "optional" directive (as explained above) indicates that the literal value associated with a token in an abstract composition tree is not to be initialized with the default literal value; finally, a "constant" directive (also explained above) indicates that the literal value associated with a token in an abstract composition tree is unchangeable (i.e., always identical to the default literal value, if present).
3) Each token production may not utilize the in-rule whitespace directives, since a token can never be broken by whitespace (by the definition of a token).

Though the definitions of token productions and structure productions are similar, there are some important differences between the two languages:

1) Each definition of a token grammar group can specify any of the composition directives, whereas the definitions of a structure grammar group may not specify any such directives.
2) Each definition of a token grammar group may not specify in-rule whitespace directives, whereas the definitions of a structure grammar group may do so.
3) Each definition of a token grammar group, as already specified, may not recursively reference itself, whereas the definitions of a structure grammar group may do so.

Note that, unlike the message and structure specification languages, this language does *not* allow the definitions of tokens to be more powerful than what can be expressed by a regular language.

5. Realization Issues

An implementing system must utilize a set of specification languages that fulfill the specification languages' requirements. However, recognizing the validity of protocol specifications written using such a set of languages is merely the easiest aspect of the function of an implementing system.

A protocol specification is defined within the construct of an understanding of how that specification will be utilized to permit communication via the specified protocol in a dynamic, evolvable manner. The various aspects include tracking the state of interactions defined by the specification, parsing/composing messages defined by the specification, translation as a whole (which includes the three fundamental actions of interaction tracking, parsing, and composing), generation of components capable of translation, and the negotiation of protocols in use, in order to determine what should be generated.

5.1. Interaction

Each interaction specification conforms to a deterministic finite automaton, which means that tracking the state of an interaction is relatively simplistic. While we will not discuss the well-understood theoretical grounding of a DFA (q.v., [Sip97]),

there are some issues that are specific to the description of a protocol as it relates to message coordination:

1) Sending or receiving. There may be situations in which a given state of the interaction may allow either an outgoing or an incoming message; in such cases, a choice of which action to take must be made.
2) Connections. An interaction can potentially span multiple connections, which imposes some additional coordination difficulties.
3) Interaction labels. An interaction that spans multiple connections must be identifiable across all connections.

While the choice of whether to send or receive a message can be problematic, the primary difficulty of dealing with interactions is the introduction of sequential (or even parallel) connections and the mechanisms required for proper coordination, so most of the discussion will be focused in the corresponding subsections.

5.1.1. Sending or Receiving

Consider the DFA representation of an interaction from the perspective of role `A`, shown as Figure 23:

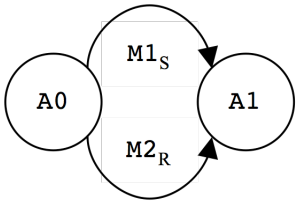

Figure 23: Interaction DFA with multiple outgoing arcs

Note that this automaton is still deterministic; even were the two outgoing arcs of `A0` to be labeled with the same message, the subscript (indicating the sending of

`M1` and the receiving of `M2`) differentiates the arcs. Herein lies our difficulty, however: How do we determine, when in any given interaction state, whether a message is to be sent or received?

If one of the arcs of the figure were to be eliminated, leaving only a single arc, the choice would be obvious. When a state has several outgoing arcs, each subscripted as a message sent, the choice is also obvious (indicating an abstract composition tree that can compose one of several different messages, discussed in Section 5.3.4). Likewise, when a state has several outgoing arcs, each subscripted as a message received, the choice is again obvious (indicating a *k* lookahead parse, discussed in Section 5.2.2). The problem lies solely with the situation in which a state has several outgoing arcs, and a mix of sent and received messages are indicated.

In such a situation, we will determine whether a message is to be composed or parsed based purely on the availability of such a message. If the abstract composition tree can form a complete message, a composition (i.e., a sent message) is in order; if data is available for reading from the connection, a parse (i.e., a received message) is in order. In the case where a message is on the wire *and* the composition tree has a completed message, the choice is essentially arbitrary; we will default to parsing in this case.

The other situation that might arise is when a message can neither be sent nor received after a given timeout; in this case, the connection should be closed, an action that may invalidate the interaction as a whole.

5.1.2. Connections

An interaction need not occur over a single connection, as with HTTP [FGM+99]. Many other protocols, such as SMTP [Kle01] and FTP [PR85], use (or potentially use) multiple connections between two entities in order to fulfill an interaction. If multiple connections are used, there are essentially two situations that can occur: connections that are used in sequence, or connections that are used in parallel. (Of course, these possibilities can both occur within the same interaction, potentially over the lifetime of a single connection, but we will consider the cases separately for the purpose of analysis.)

Sequential connections are the easier of the two scenarios. This occurs when one connection is used to send messages as part of an interaction, the connection is closed or otherwise abandoned, and then a second connection is used to exchange further messages of the same interaction. For example, SMTP specifies such a multi-connection interaction when dealing with bounced mail messages; the original message is sent from one mail server to another over a first connection, and the bounce message is sent back over a second connection (possibly at a much later time) [Kle01]. To accommodate this, we must provide a mechanism (described in the next subsection) to correlate the separate connections, in order to obtain a reference to the relevant interaction used by both connections.

Parallel connections are more difficult to handle. Parallel connections occur when two connections are used simultaneously to send messages as part of the same interaction. For example, FTP specifies concurrent connections, one for the coordination of file transfer (the “control connection”) and one for the transfer itself

(the “data connection”) [PR85]. However, this requires concurrent traversal within the DFA of the interaction, but since our interactions are deterministic, such parallelism is impossible in the general case. Therefore, we preclude parallel connections for the time being, thus eliminating the class of protocols that follow a format similar to FTP. It is possible that language features can be later added in order to accommodate such a feature, though it will likely merely involve marking points of exclusive access in the interaction DFA.

5.1.3. Interaction Labels

Multiple (sequential) connections require us to provide a mechanism that correlates two (or more) connections used by the same interaction. (Note that, for now, we assume that a connection is specific to a single interaction, though an interaction is not necessarily specific to a single connection.) This mechanism is one of labeling the messages as specific to an interaction.

Each label is considered unique to an interaction. A label may be a string, a set or list of strings, or any other identifier that is determinably unique. The specification languages developed for this work allow us to annotate a token as part of the interaction label, and the interaction label is then the sequence of literal values associated with such tokens as a message is either composed or parsed. This label can then be used as a reference for the relevant interaction.

An empty label is possible, provided all communication between two roles takes place over a single connection; HTTP adopts this strategy [FGM+99]. Otherwise, an interaction label is our mechanism to relate different connections as communicating messages of the same interaction; the “Message-ID” header of an

SMTP message is an example of this [Res01]. Given these considerations, some properties can be derived with respect to the possibilities of the presence of an interaction label:

1) A message that *does not* include an interaction label must be either a subsequent message over an established connection or an initial message between two roles that will use only a single connection during the interaction.
2) A message that *does* include an interaction label that *is not* associated with a current interaction must be the first message exchanged over a first connection between two roles during an interaction. The new label and the new interaction (and their association) must be recorded for future reference.
3) A message that *does* include an interaction label that *is* associated with a current interaction must be either the first message exchanged over a subsequent connection between two roles during the interaction or a subsequent message exchanged over a connection between two roles during the interaction.

The second possibility of the third property introduces the interesting question of how a message should be handled if it is not the first message of a given connection and its (possibly empty) interaction label does not match the interaction label of the connection's first message. This is not technically a parsing error, given that the message format is indeed correct, but rather some semantic element of the message is inconsistent. Assuming we treat this scenario as an error rather than

ignoring the interaction label of all subsequent messages communicated over a connection, this is the only situation in which the automated translation system as a whole is concerned with the semantics of a message.

5.2. Parsing

Issues with respect to parsing have long been known and studied (e.g., [Knu04]). Consequently, we will not cover details such as the reduction of left-recursive rules in an LR grammar in order to attempt an LL grammar. However, there are still some specific issues that relate to the specification languages and their features, both provided and required:

1) Tokens versus structures. The question of whether a grammatical element should be considered a token or a structure is determined by many factors, including the expressiveness required, other syntactic requirements, and the subjective judgment of the developer who is specifying the protocol.
2) Multiple messages and *k* lookahead. More than one different message may possibly be received by a particular interaction role when the current interaction is in any given interaction state. While each message necessarily (by requirement) is limited by LL(1) expressivity, the initial token or tokens that determine which message should be recognized may require *k* lookahead, where k is greater than one.
3) Whitespace. Different parts of a message may use a different whitespace specifier (or none at all).
4) Context-sensitive termination. Of the four methods of terminating a repeated grammar element, two can be accommodated by context-free

grammars, and two cannot (practically speaking). However, one of the latter pair is commonly used and so we must provide a mechanism to specify and handle this termination when parsing a legal protocol message.

5) Error handling. While the composition activity handles an error such as a superfluous token by simply not including it in the message, and as a consequence does not construct what would otherwise be an inconsistent message, the parsing activity must deal with errors in a more immediate fashion.

5.2.1. Tokens Versus Structures

A token can be interpreted as the smallest unit of meaningful information. This is a bit of a misleading description, however, since a token can itself be comprised of other tokens. Presumably, the subtokens of such a complex token are meaningful in the same way as is the token, which makes the token not the "smallest" unit of meaningful information.

Luckily, the term "meaningful" is itself subjective, and one can interpret meaning on several different levels. Even so, subtokens may have meaning in the same way as a top-level token, at the discretion of the developer. We will consider any token, regardless of whether it is used in the definition of a structure or another token, as having some measure of meaning if associated with a concept. Thus, any token can be interpreted as representing meaningful data, and any definition (be it message, structure, or token) that references tokens or compositions of tokens can be interpreted as representing a relationship among the relevant data; messages and

structures always represent relations among data, and tokens can represent relations among data or the data itself (perhaps both).

Of course, every token that expresses a relationship among other tokens could be represented as a structure (unless the token also represents data), since the LL expressiveness of structures is more powerful than the regular expressiveness of tokens. Given that, it is recommended that a grammatical element that is intended solely to represent a relation (composition) of data without itself being considered data should be expressed as a structure, rather than a token (even though it might be expressible as a token).

In addition to expressivity and the conceptual perceptions suggested above, other syntactic considerations will also influence whether a composition of tokens is represented as a structure or a token. One such issue is whitespace (discussed in more detail below). A token represents a definite pattern, and arbitrary whitespace is thus not permitted to occur between the components that comprise the token. A structure, on the other hand, might allow whitespace between the elements being related. Therefore, if whitespace is to be allowed, the grammatical element must be defined as a structure, rather than a token.

To illustrate some of these principles, consider the following message definition (used throughout this chapter, to illustrate various issues of both parsing and composition), broken into the message definition itself (`M`), structure definitions (`S`), token definitions (`T`), and subtoken definitions (`K`):

```
M (S1 S4)
```

```
S1 (S2 S3)
S2 (<T1> <T2>)
S3 (<T3>)?
S4 (<T4> <T5>)+

T1 ("D1") = C1
T2 ("D2") = C2
T3 (<K1> <K2>) = C3
T4 ("D4" | "V4") = C4
T5 (<K3>) = C5

K1 ("D3")
K2 ("V3")
K3 ("D5")
```

Note two of the tokens are defined in terms of subtokens, yet they are associated with concepts, making them important data in their own right. Other of the tokens (noting `T4`, in particular) are defined in terms of literal values, even though they could be defined in terms of subtokens that represent the individual literal values. (Presumably, the literal values have no meaning in isolation.) Structures are defining relations among the tokens, even if the relation is trivial (e.g., `S3`).

This is, of course, a contrived example for the purpose of illustration, but these ideas are represented in realistic scenarios. The HTTP 1.0 version specifier, "`HTTP/1.0`", can be viewed as an item of data itself or as a relation of the data "`HTTP`", "`1`", and "`0`". Concepts may be associated with any or all of these elements. The version specifier, in turn, can be perceived as a component of an HTTP request line; if the request line is not viewed as a token (even though its structure is regular), it will be defined as a structure that relates the version specifier to other elements of the request line.

From this discussion we can extract some guidelines for determining when a grammatical element should be defined as a token or a structure:

1) If the text associated with the element is referenced by the definition of a token (i.e., if the definition of the element includes string or character literals), the element must be defined as a token.
2) If the text associated with the element is meaningful as data, the element should be defined as a token; otherwise, the element should be defined as a structure.
3) If whitespace is permitted between the components of the element, the element must be defined as a structure.
4) If the expressiveness required by the element is not regular, the element must be defined as a structure (presuming the element can be described by an LL(1) language).

If, within any intended message specification, two or more of these guidelines are in contradiction (e.g., an element of data that allows arbitrary whitespace between the elements that compose the data), this may indicate an unnecessarily complex token, though it certainly describes elements that are not appropriately described by the principles provided by this work.

5.2.2. Multiple Messages and *k* Lookahead

All messages are required to have a structure describable as an LL(1) language (with the exception of context-sensitive termination). Consequently, any top-down parser should be able to handle any message described by the specification languages.

This presumes that the message being parsed is known, however. Before an interaction is begun (if more than one interaction is possible) or if more than one

message can be received (by the application component implementing the interaction role in question) during the current interaction state, the message may not be determinable by simply reading the first symbol.

For example, the status line of an HTTP response messages begins with the HTTP version specifier, and is followed by a space, the status code (e.g., "`200`"), another space, and a status message (e.g., "`OK`"). Assuming that the status line is represented as a structure rather than a token, and given that the status code is consequently the token that distinguishes the message, it is impossible to determine which response is to be parsed unless more than one token (the version specifier, the separator, and the status code) are read beforehand.

This requires that the lookahead at the beginning of the parse be at least the smallest number such that the determining token or tokens of each message that can possibly be received in the given context be reachable. For the HTTP example, this value would be three, assuming the version specifier is defined as a token.

Note that, for a top-down parser (which is a logical choice given our restriction of expressivity), this places further restriction on the specification of a message. The token or tokens that uniquely identify one message from all others that can possibly be received in the same context must be reachable within a definite (preferably small) number of tokens. This restriction prohibits message specifications that allow arbitrary repetition prior to the relevant, identifying tokens. (Bottom-up parsers, on the other hand, may be able to handle such a scenario without a multi-symbol lookahead, though the choice of a bottom-up parser, while not contradictory, is not an obvious choice of utility.)

The precise lookahead for the beginning of a parse is left as an implementation detail; it may be static or dynamically determined from the given message specifications. An alternative to this violation of an LL(1) specification involves the idea of message hierarchies, which will be researched as part of future work.

5.2.3. Whitespace

For our purposes, whitespace is any token filtered by the lexer as irrelevant to the parser (regardless of whether the token is composed of "space"). Whitespace is important for the purpose of token separation, something that will be discussed in greater detail in Section 5.3.9.

Because the specification of a protocol is intentionally piecemeal, the whitespace permissible within one structure may be different from that permissible within another, or even unknown. The parsing activity must be capable of switching contexts in the sense that different whitespace tokens may appear at different points within a message.

5.2.4. Context-Sensitive Termination

Parsing must handle our special case of repeated grammar element termination, in which one grammar element in an earlier part of the message indicates how many of another grammar element occur in a repetition later in the message. Parsing such a message is relatively simple, though as we shall see later the composition activity is considerably more difficult.

Given that a counter token occurs before the relevant repeated (counted) grammar element, we associate an integer with each token used as a counter. When

the counter is parsed as a token, we convert the literal value and store it as the associated integer. Then, when the repeated grammar element is encountered, we simply count the number of repetitions and terminate the repetition when it reaches our stored integer.

A potential problem exists when more than one instance of the counter token is present in the message before the counted grammar element. In such a case, we can simply override the old associated integer value. Alternately, we could specify which of the values to use (though our approach uses the former technique). Regardless, the parsing of counted, context-sensitive grammar elements amounts to counting the repetitions, comparing the count to the integer interpretation of the literal value of the counter token, and instructing the lexer to move on to the next symbol once the bounds of repetition have been reached.

This does introduce another difficulty that involves nested or sequential repetitions of the same grammar element. For a context-free specification, simplifiers such as the greedy, reluctant, and possessive quantifier rules of the POSIX regular expression specification [IEEE04] can be used, but they are more difficult to apply (and in a few cases impossible to apply, due to ambiguity of rule application) when the boundaries are essentially unknown, as is the case with context-sensitive termination. Therefore, we impose the restriction that a counted rule either end the definition of a production or be immediately followed by a non-optional rule or a *different* token or structure whose principal rule is non-optional. For example, consider the following structure and token definitions:

```
S (<N> (<T>)* <T>)
```

```
N ("0" | ["1"-"9"]("0"-"9")*)
T ("A")
```

The token `T` in the structure can be repeated zero or more times, according to the application of the Kleene star, followed by a single instance of `T`. This is identical to `T` repeating one or more times, and results in an equivalent rule for `S`:

```
S (<N> (<T>)+)
```

Consider, however, the repetition pattern for `T` being specified in terms of the counter token:

```
S (<N> (<T>){0, <N>} <T>)
```

This rule is more difficult to automatically simplify unless we allow cumulative expressions as repetition boundaries. In the absence of such, the trailing instance of `T` requires backtracking during the parse in the case where the repeated instances of `T` are not exactly `N` in number. (Note that this example situation can be avoided by simply moving the trailing instance of `T` in front of the repeated instance of `T`.)

5.2.5. Error Handling

Errors that occur during parsing are different from errors that might occur during composition, because the error is occurring at a point when there is no obvious solution as to how to handle the situation. An error that occurs during composition is handled by simply composing a different message, or not composing a message and instead letting the interaction fail. At this point, we consider an error during parsing to be a “fatal” error in the sense that no recovery is possible, given that the specification languages do not accommodate the association of an appropriate action with a

grammar element when the grammar element was not parsed as expected. (In some cases, it might be possible to direct recovery from an invalid parse.)

Note that, if the composer of a translator that is sending a message performs correctly, the parser of a receiving translator will not encounter a situation in which the message is incorrect and causes a parsing error (barring extremes such as transmission error). However, not every message received need be composed by a system conforming to this work; messages could be composed directly by the application or even by hand, independent of the protocol specification, in which case errors of composition (or at least inconsistencies with respect to the specification) may occur. Thus, parsing must at least consider the possibility of errors, though the strategy for dealing with them is primitive.

5.3. Composition

Composition is a complex activity, involving several aspects of the specification languages that allow us to define some of the special behaviors of a protocol specification, such as context-sensitive repetition. This section will examine the composition process in detail.

Figure 24 shows the subtree of the abstract composition tree (ACT) that represents the message production of our simple example grammar. The triangular shape at the root represents the message production, the square shapes represent the structure productions, and the hexagonal shapes represent the token productions, which are annotated with the concept to which each has an association. Note that an ACT is unconcerned with concepts to which the structure productions or the message production may be associated; only concepts to which token productions have an

association are relevant, since they will be used in the process of "matching" concepts provided by an application.

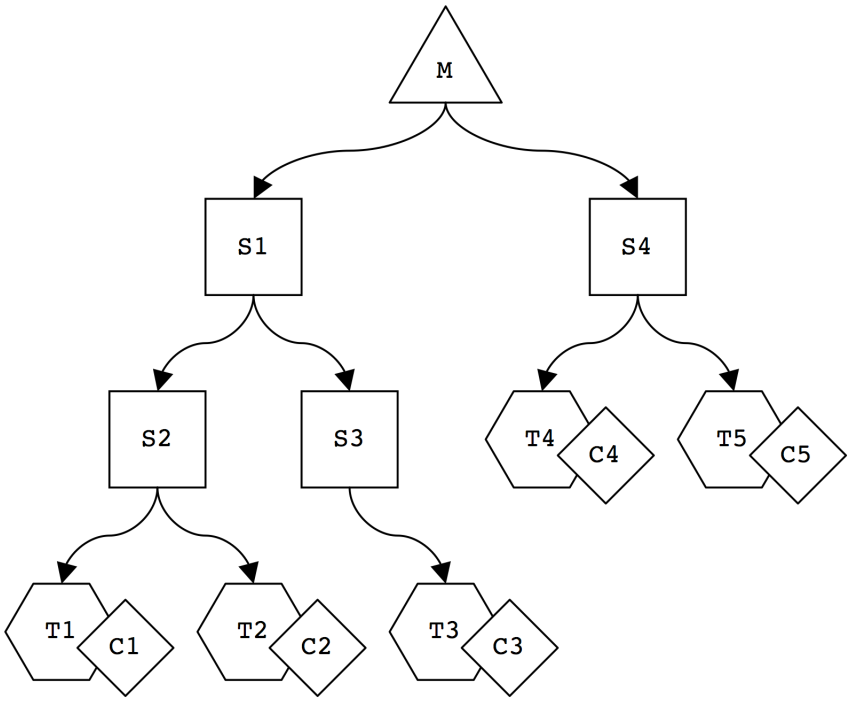

Figure 24: Condensed abstract composition tree

Because the example grammar is extremely simple, so too is the corresponding subtree of the ACT; complexity will be added to the tree as issues of composition are discussed. These issues include:

1) Alternating levels of composition. Each message subtree of an ACT (and even the ACT as a whole) can be divided into alternating levels of choice and concatenation.
2) Patterns of repetition. Optional and repeated rules introduce special consideration for message composition.

3) Latent construction. In the general case, the structure of a subtree of an ACT that represents a message cannot be statically determined, and must be dynamically constructed as concepts are provided by the application.
4) Multiple message possibilities. In the general case, an ACT represents not just a single type of message, but several possible types of messages, any one of which can be sent by a particular role during a particular state of the relevant interaction.
5) Literal values associated with token concepts. A concept does not specify the literal value of a production related to the concept; therefore, an associated string must be provided with a token concept (generally speaking) when one is added to an ACT.
6) Default literal values associated with tokens. Given that applications supposedly have little to no knowledge regarding the structure of the ACT being constructed (or even which protocol message will ultimately be composed), it is unreasonable to expect that, in the general case, an application will be able to provide information sufficient to complete an ACT in the sense that a valid protocol message can be composed. Consequently, a mechanism of default literal values to which tokens can be associated is used to further the "completeness" of an ACT.
7) Context-sensitive termination. Of the four methods of terminating a repeated grammar element, two can be accommodated by context-free grammars, and (in practice) two cannot. However, one of the latter pair is commonly used and so we must provide a mechanism to specify and

handle this termination within the construction of the ACT and the composition of a legal protocol message from the ACT.

8) Attempts to compose a message from the ACT. Once an ACT is constructed (after, presumably, the application has finished adding concepts and associated literal values), a message must be extracted from the tree.
9) Whitespace. A message may be specified as freeform for reasons that may include readability, ease of handcrafting messages, or even ensuring proper separation between tokens that otherwise might be ambiguously or improperly parsed. Regardless, the specification languages must provide features to deal with whitespace not only for parsing but also for composition.

Each of these issues will be individually discussed in the following subsections, adding to our basic figure as appropriate. Following this discussion, some possible variants of the construction and composition algorithms (presented in detail in Appendix C) will be introduced.

5.3.1. Alternating Levels of Composition

The diagram of Figure 24, as mentioned, is simplistic as a reflection of the grammar that describes it. This diagram is far from the whole story, however, even for this simple case. We can construct a static diagram to represent even an abstract version of `M` because the structure of `M` is definite. There are no options with respect to its structure; if there were, this subtree of the ACT could not be constructed in definite form.

The fundamental notions of composition we must handle involve *choice* and *concatenation*. The figure in question reflects only concatenation, because only concatenation is explicitly specified by the grammar. The lack of choice in the grammar is technically an illusion, however; in fact, several choices (of one possibility, without delving into the semantics of "is one really a choice?") exist in the grammar. Let us re-examine the simple production of `S1`:

```
S1 (S2 S3)
```

It is clear from the production that `S1` is the concatenation of `S2` and `S3`. What is not so obvious is that the concatenation is a choice, albeit the only one. The choice becomes more obvious with two or more, so for the sake of clarity we will invent a production, `X1`, similar to `S1`, except that we will add a second alternate (that references an imaginary structure production, `A`) to the principal rule:

```
X1 (S2 S3 | A)
```

Here there are two choices to be made within the primary rule that defines the production, versus the one of `S1`. We will invent another, extending production in order to illustrate the nesting capabilities of choices and concatenation:

```
X2 (S2 S3 | (A | B))
```

In fact, productions can (within the limits of expressivity) be defined using essentially arbitrary nestings of rules (concatenation) and alternates (choice):

```
X3 (C (D | E) | (S2 S3 | ((A D (E | F) G) | B)) | H)
```

No matter how complex the definition of the production, it can always be viewed as alternating concatenation and choices. (Note that these "`X`" rules are for the use of illustration only, and are not part of our sample grammar.) Each rule (identifiable in the productions by a corresponding pair of parentheses) identifies a

(technically unordered) set of alternates (possibly only one), and each alternate identifies a set of (ordered) subproductions to be concatenated, any of which may itself be a rule. This defines a recursion of alternating levels. This recursion is terminated within the definition of a production by alternate subproductions that are either literals or references to other productions, rather than rules. More specifically, for the definition of a production that is not self-referencing, the recursion is ultimately terminated by literals; for the definition of a production that is self-referencing (allowable only for the definition of structure productions), the principal rule of the definition must have an alternate that does not ultimately reference the production (which is in any case a condition of the limited expressivity allowable for production definitions).

Figure 25 shows the expanded version of Figure 24 (using the grammar of page 91, for now ignoring the repetition patterns for rules), in which the rules and alternates of the message production and structure productions have been made explicit. Rules are shown as a shape with a solid line, with the principal ("definition") rule of a production given a shadow. (Note that, since no production definitions of our simple grammar specify subrules, no rule shapes lack a shadow.) Alternates are shown as a shape with a broken line. Notice that the entire tree can be divided into peer, alternating levels of choice and concatenation.

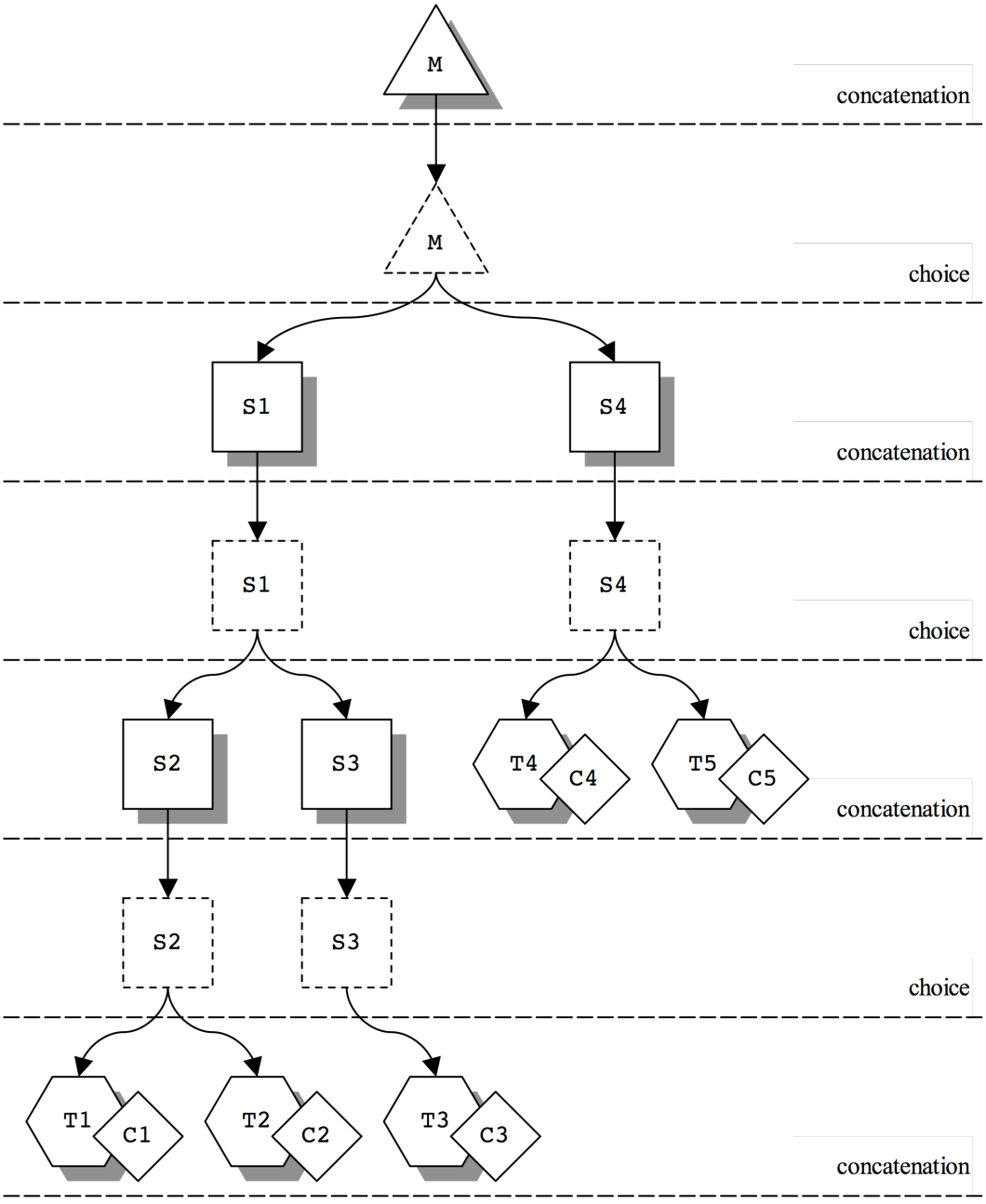

Figure 25: Expanded ACT with explicit rules and alternates

The choice and concatenation levels stretch horizontally across the tree as an aspect of the following criteria, paraphrased from above: Each rule specifies one or more (unordered) choice nodes, among which exactly one is selected per repetition

during composition, and each alternate specifies one or more (ordered) concatenation nodes, all of which are sequentially connected during composition; further, an alternate is a type of choice node (the only type, actually), and a rule is a type of concatenation node. Our ACT is thus a derivative of an "and-or" tree, isomorphic to Horn clauses [Hor51].

5.3.2. Patterns of Repetition

An important detail of the above criteria may have escaped attention; it is emphasized in the following: "Each rule specifies one or more choice nodes, among which exactly one is selected *per repetition*." This is an important detail, because it alludes to our ability to define rules as repeating arbitrarily.

Such repetition is, in practice, commonly expressed using the standard regular expression operators, though it can be equivalently expressed using (recursive) BNF rules:

regular expression	**EBNF**	**BNF**
`P := R?`	`<P> := [ <R> ]`	`<P> := | <R>`
`P := R*`	`<P> := { <R> }`	`<P> := | <P> <R>`
`P := R+`	`<P> := <R> { <R> }`	`<P> := <R> | <P> <R>`

(In the specification languages developed for this work, both forms of expression, definition via regular expression operators or definition via a recursive BNF derivative, are allowed for structure productions, but only the former is allowed for token productions.) Rather than convert all such repetitive rules into (possibly multiple) BNF productions, we will consider the common patterns of repetition directly within the ACT.

To illustrate this issue, we will use the definitions of `S3` and `S4`:

```
S3 (<T3>)?
S4 (<T4> <T5>)+
```

This consideration of rule repetition patterns is shown in Figure 26. Notice that the grammar is no longer definite, given that one rule is optional and another can be repeated an arbitrary number of times (but at least once); this arbitrariness is reflected in the uncertainty implied by the new diagram notation. The dotted box (previously solid) that represents the primary rule of the definition for the structure of `S3` indicates that this rule is optional. The multi-lined box (previously singular) that represents the primary rule of the definition for the structure of `S4` indicates that this rule is repeatable an arbitrary number of times (one or more).

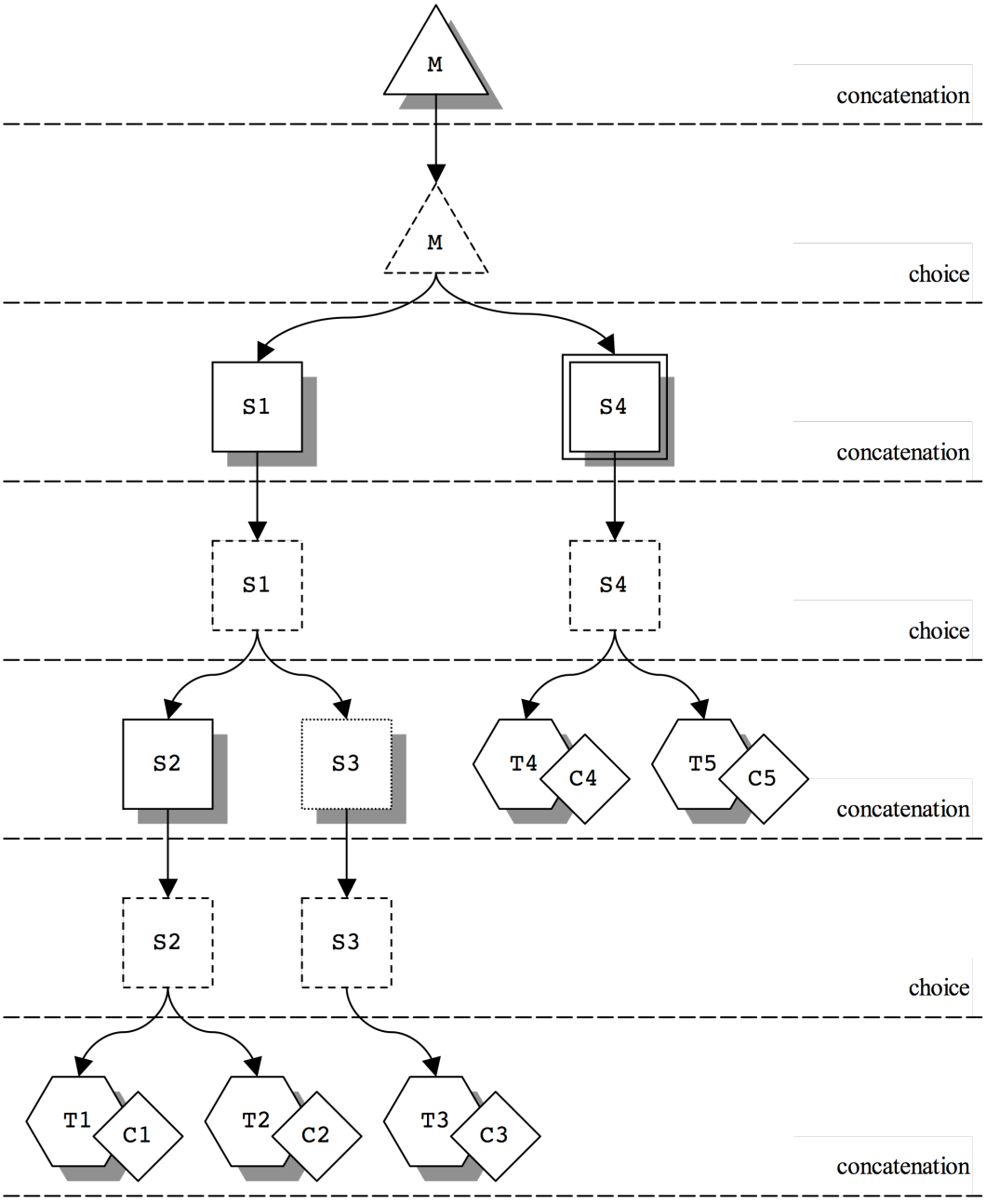

Figure 26: Expanded ACT with complex repetition patterns

The problem introduced by the new indefinite structure of the tree is that the form of a subtree of an ACT that represents a message production can no longer be predetermined, which, by implication for practice, means that the ACT must be

dynamically constructed as new information is presented. (It is important to note that the same issues arise when rules are allowed empty definitions or when rules are specified recursively, so a BNF-like equivalent to a regular expression operator would not solve the problem of ambiguity due to multiple potentialities in the ACT.)

The four repetition patterns that have been discussed (three that correspond to the standard regular expressions operators and one that corresponds to the lack of a regular expression operator) essentially describe only the endpoints of a range of possibilities:

operator	optional	repeatable	lower bound	upper bound
(none)	no	no	1	1
?	yes	no	0	1
+	no	yes	1	(unlimited)
*	yes	yes	0	(unlimited)

Notice the limited possibilities expressed by the lower bound and the upper bound; theoretically, the lower bound can be any value in the range [0, upper bound], and the upper bound can be any value in the range [lower bound, ∞). It is possible to allow for other, more particular boundary conditions to be specified. (In fact, this will be a requirement to deal with context-sensitive termination.) For our purposes, a rule will be considered *optional* if the lower bound of its repetition pattern is 0, and a rule will be considered *repeatable* if the upper bound of its repetition pattern is greater than 1.

5.3.3. Latent Construction

Given the possible repetition patterns of rules (and possible recursive definition of structure productions), each subtree of an ACT that represents a message describes not a single tree but rather a family of trees. Consequently, such a subtree

cannot be statically created prior to the information necessary for its construction being provided (i.e., it must be created dynamically). Therefore, we will refer to the construction of an ACT as being "latent" in the sense that references to productions within the definition of a production will be represented by "placeholders" in the corresponding nodes of the ACT until the construction of such is needed.

Figure 27 shows the subtree of the ACT that represents `M` as it would appear prior to any information being added to the tree. The "star" shapes represent placeholders for the productions referenced by the primary rule that defines `M`. (It is also possible to simply have a placeholder for M, rather than to expand its principal rule, but that diagram is not only uninteresting but also nondescript and therefore useless for the purpose of illustration.)

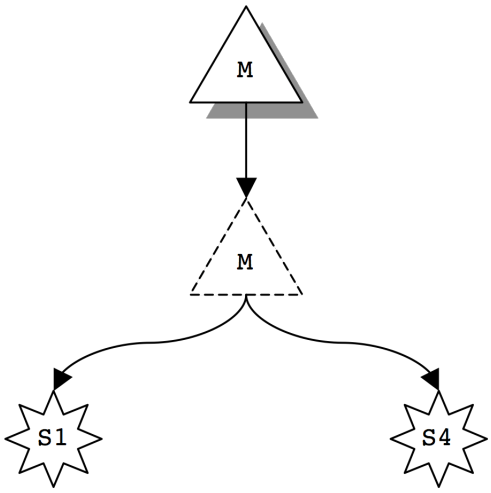

Figure 27: ACT prior to latent construction

Now we can add information to the tree in the form of concepts. Suppose we want to add concept `C2` to the tree. Figure 28 again shows the state of the initial tree

as this first concept is added. Note that the concept is added (from above, explained in the next subsection) to the *root* of the subtree that represents the message.

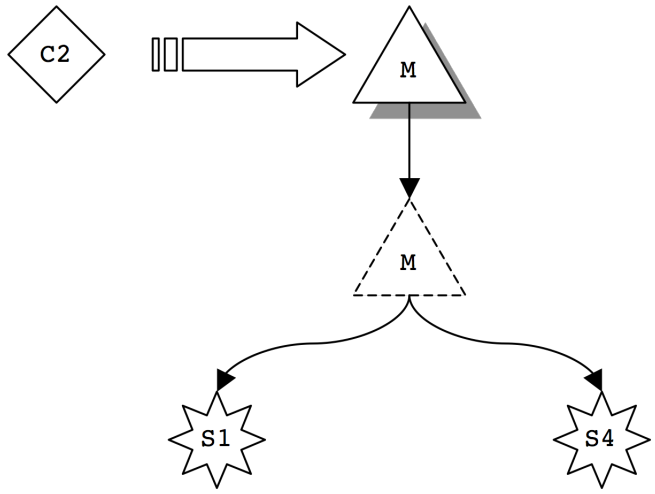

Figure 28: ACT at start of latent construction

The concept is passed down the tree using a preorder traversal. When passed to the placeholder for `S1`, the corresponding subtree is constructed dynamically, again with nodes of the subtree that represent references to productions being occupied by placeholders. Figure 29 shows the state of the tree after the subtree corresponding to the definition of `S1` has been constructed.

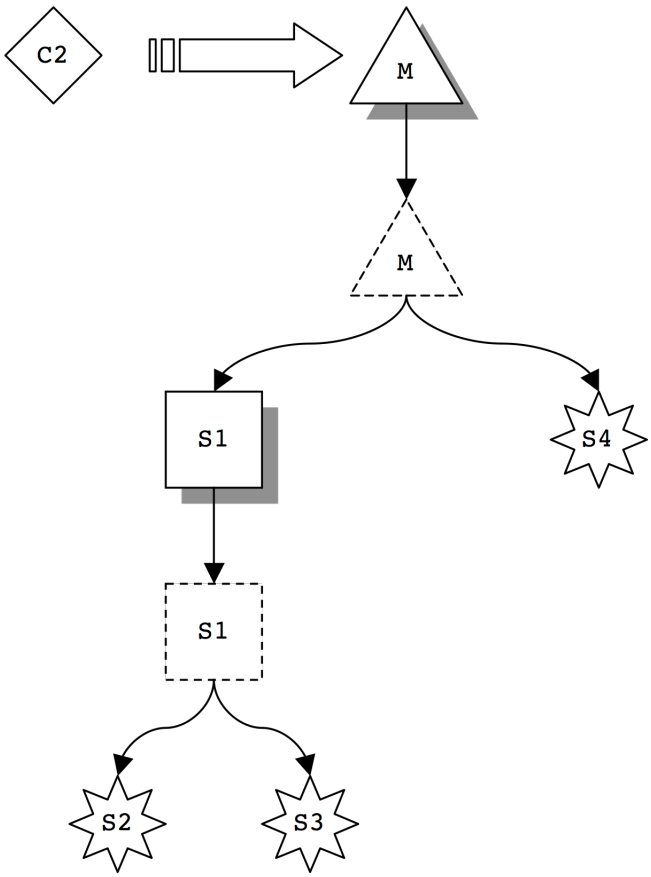

Figure 29: ACT latent construction of `S1`

This process of dynamic construction is repeated as the concept is passed to the placeholder for `S2`. Figure 30 shows the next phase of the tree's structure.

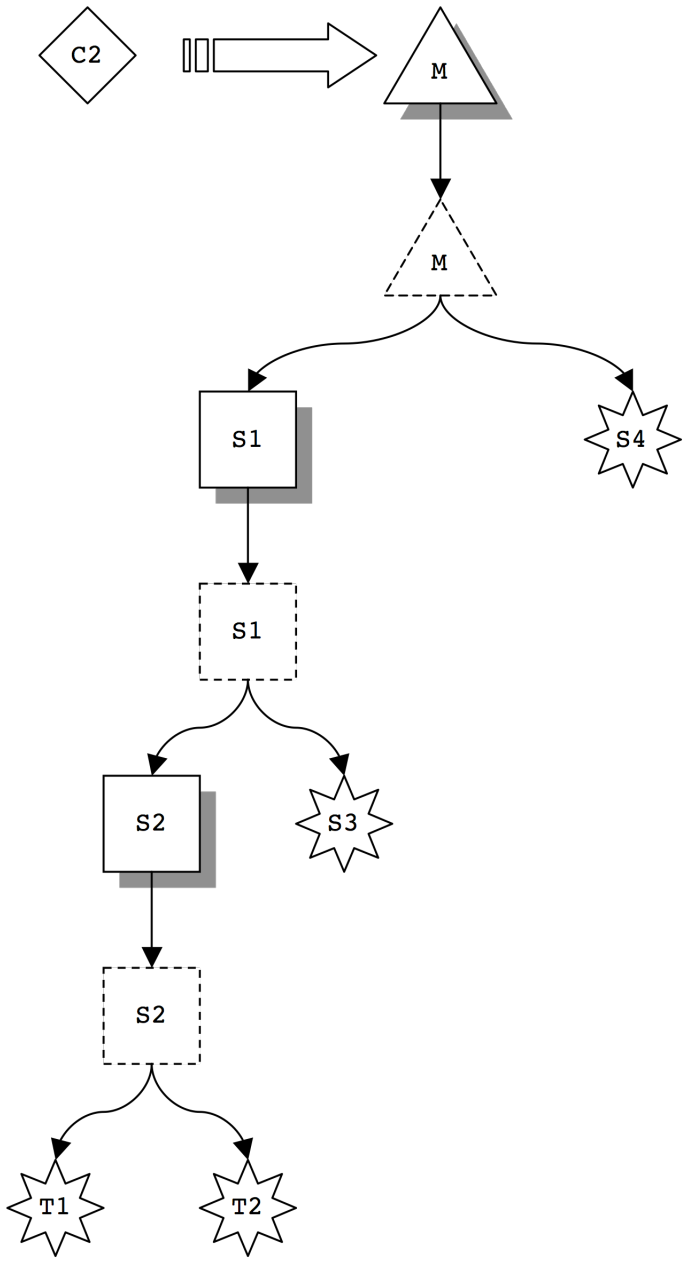

Figure 30: ACT latent construction of `S2`

Now the concept is passed to the placeholder for `T1`, which represents a token production. The corresponding subtree is constructed in the same way, though for brevity of the diagram the details of the definition of `T1` are not shown. Instead, we

simply identify that the concept to which `T1` is associated (`C1`) does not match the concept being passed around the tree; thus, the diamond of `C1`, as an attribute of `T1`, will for now remain unshaded, indicating an unmatched concept. Since the concept added to the ACT has not yet been matched, it is passed to the placeholder for `T2`, where a construction identical to that of `T1` takes place. This time, however, the concept is matched, as indicated by the shaded diamond in Figure 31.

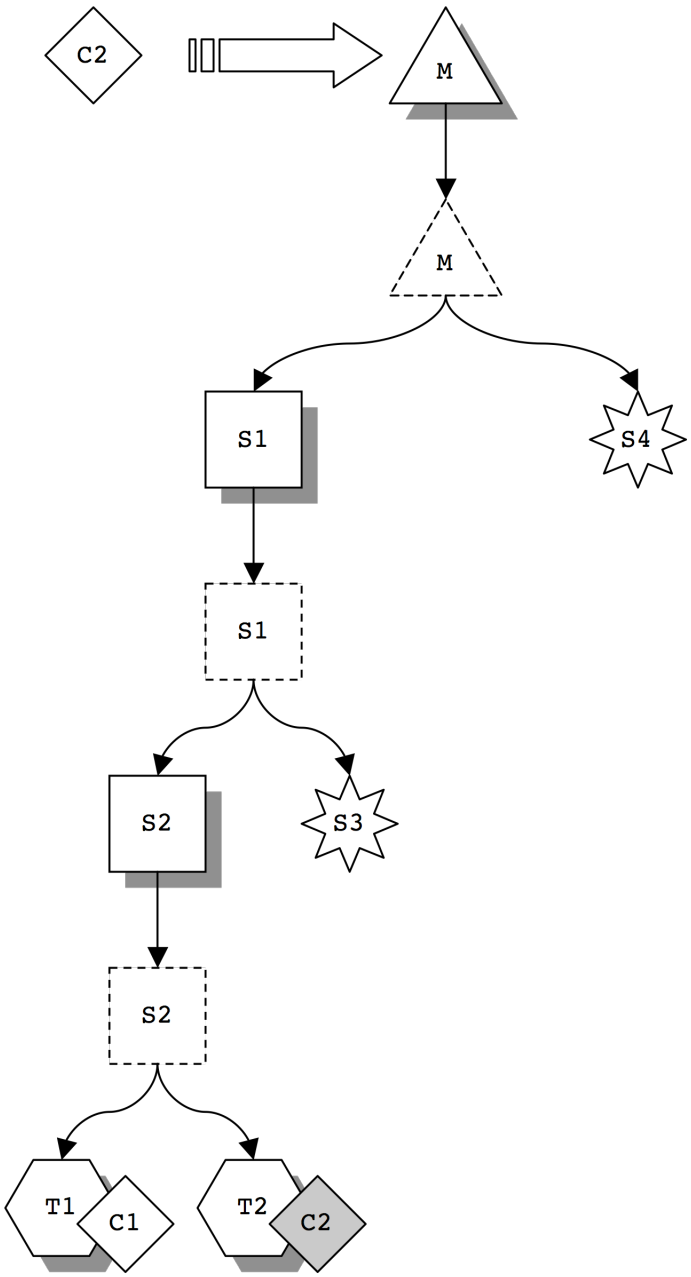

Figure 31: Match of concept C2

Note that placeholders to the “left” of the right-most token node with a matched concept are resolved as subtrees, and placeholders to the “right” of the right-most node with a matched concept are never resolved. (There will be instances in

which placeholders to the left are not resolved, but those will be covered in a later section.)

We have added one informational concept to the subtree of the ACT that represents `M`. This is insufficient to compose `M`, however, so more concepts must be added. Suppose that we attempt to add the concept `C4`. The same algorithm is used, and the concept is passed around the tree via a preorder traversal. Several placeholders have already been resolved, and these need not be resolved again, only tested. The node representing `T1` is tested for a match and rejected. The node representing `T2` is tested for a match and rejected, not because the match failed but because a concept matching that associated with `T2` has already been added. The traversal backtracks to the alternate node for `S1` and continues down the next subtree (which is dynamically created), and again finds no match. Finally, the traversal backtracks to the alternate node for `M` and continues down the next subtree, ultimately finding a matching concept to which `T4` is associated. The resulting relevant portion of the overall ACT is shown in Figure 32.

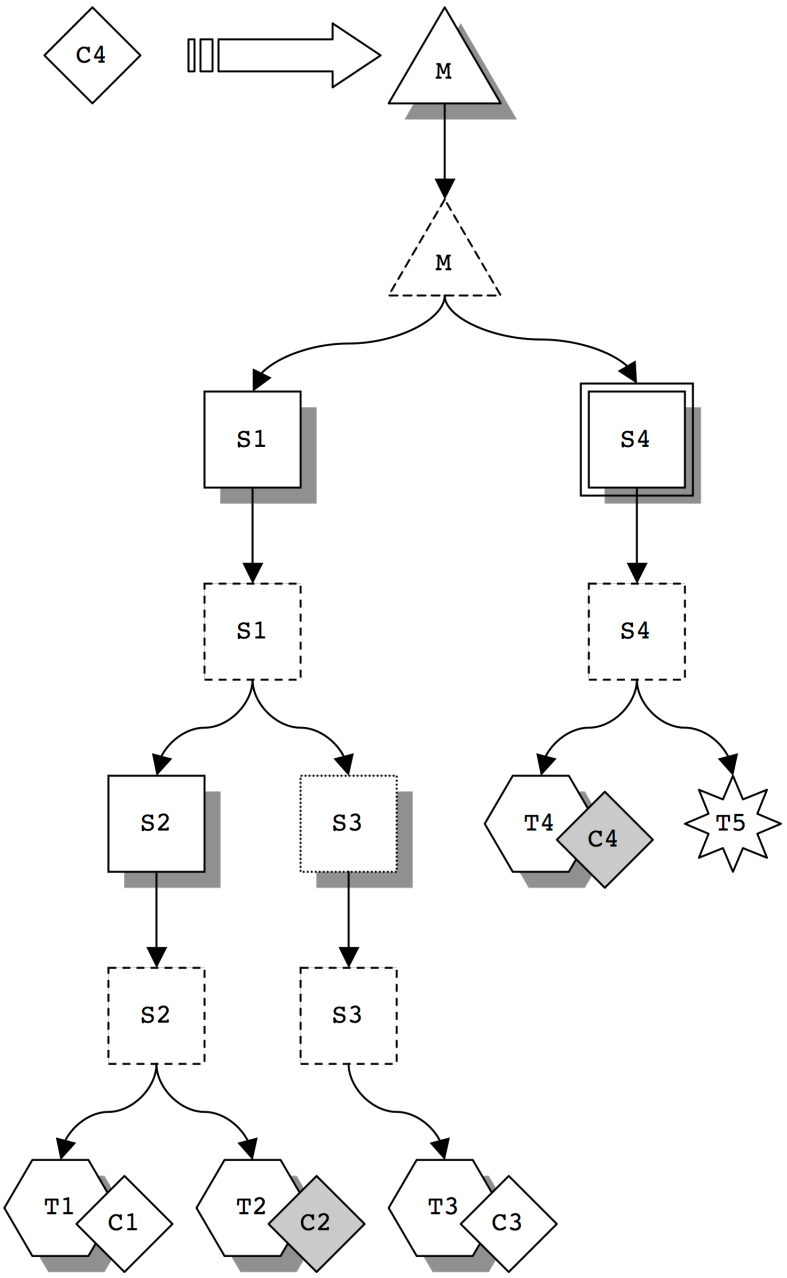

Figure 32: Match of concept C4

As was pointed out previously, however, our reversion to the original definition of S4 allows one or more repetitions of the rule. We can consequently add another instance of C4, with the results pictured in Figure 33. Notice that the first

placeholder for T5 has been resolved (by the traversal of the second instance of C4), but the second has not.

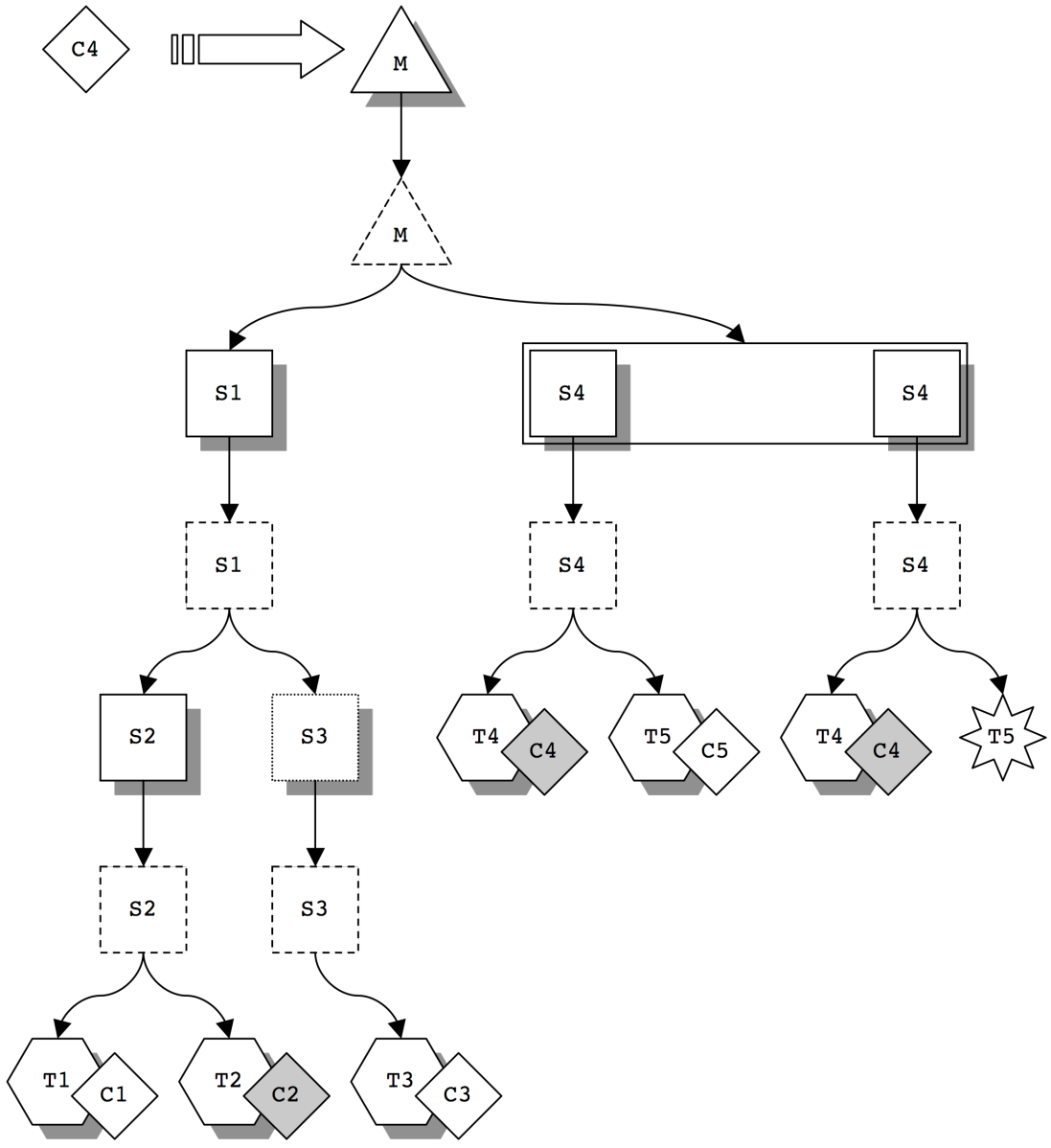

Figure 33: Second match of concept C4

The addition of two instances of C4 to our subtree of the ACT is still not sufficient to derive a message, however. Neither the (required) rule for S1 nor the

(required) rule for `S4` (in either instance) is satisfied. To complete the construction of the tree to a degree such that a message can be extracted, we will add concepts `C1` and `C5` (and a redundant instance of `C2` for a brief illustration) as shown in Figure 34.

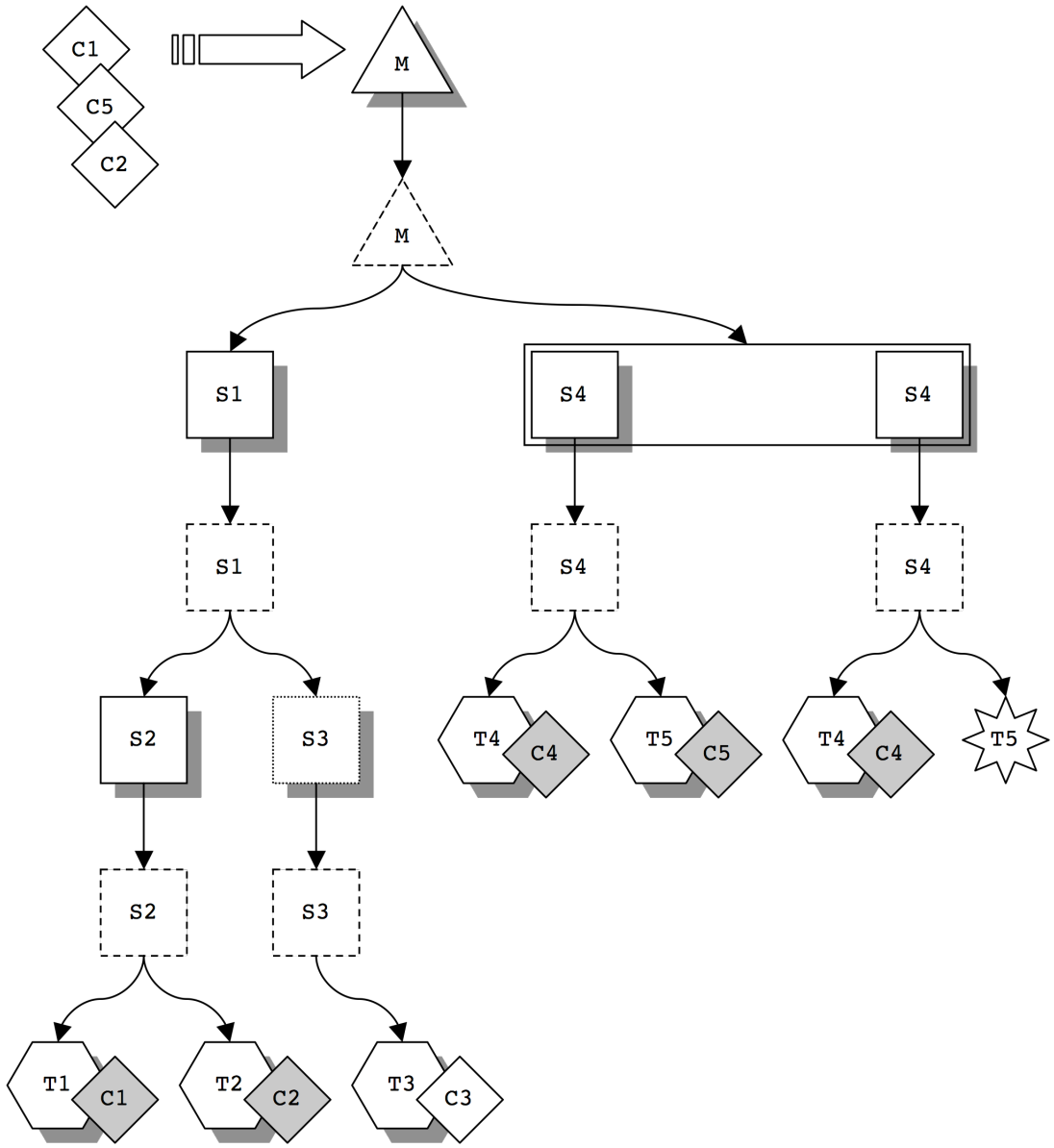

Figure 34: Match of concepts `C1` and `C5`

With the addition of `C1` and `C5`, the dynamic construction of the tree has now "filled" the tree sufficiently in order to compose a message; no sections that are not optional (simply because they have been specified as such or because they are extraneous replications) are incomplete. The composed message has the form:

```
<T1> <T2> <T4> <T5>
```

Some things to note:

1) The token `T3` is not included in the composition of the message, because it is ultimately optional and because no matching concept was provided.
2) The second instance of `T4` is not included in the composition of the message, because the subtree of which it is a part is not itself complete; the fact that this subtree is not complete does not make the composed message invalid, given that a previous instance of the subtree *is* complete and thus satisfies the "one or more" condition.
3) The redundant concept, `C2`, added to the subtree was not included in the dynamic construction of the tree, and was thus ignored; this will become more important later, when considering variants of the algorithm. Note that even though the concept would have essentially traversed the entire tree its traversal did not cause the placeholder of the second instance of `T5` to be resolved. (This, ultimately, is a kind of required optimization; see below.)
4) No rule is defined as more than one alternate. (If a rule is defined as more than one alternate, the concept must be passed down the subtree that corresponds to each alternate until matched; because the alternates are a set rather than a sequence, the traversal algorithm used is thus technically a preorder traversal only at the alternating levels of concatenation.)
5) Several optimizations are possible. For example, given that the first instance of `C4` did not match any concept in the subtree that corresponds to `S1`, the second instance of `C4` need not even be passed down this

subtree. This particular optimization will become a necessity when dealing with recursively defined structure productions, discussed in a later subsection.

Regardless of these additional issues, this subtree of the ACT is "complete" in the sense that the resultant composed message can be validly (and uniquely) parsed according to the definition of `M`.

5.3.4. Multiple Message Possibilities

Until now, we have been dealing with only a subtree of the ACT, one that describes an entire message. However, during any given state of an interaction, the interaction may specify that a role could send one of *several* possible messages. For example, an HTTP GET request issued by a client may be answered either by a 200 response or by a 404 response from the server.

In this case, the ACT must be capable of constructing any of the possible messages without knowing in advance which possibility will ultimately be completed (and thus composed). This can be accomplished by simply constructing each possible message in parallel as a separate subtree of the ACT.

Recall that the subtree representing `M` could be divided into alternating levels of concatenation and choice, the top level being concatenation. The ACT, then, can simply be viewed as a single rule with each possible message being an alternate of the rule, as shown in Figure 35.

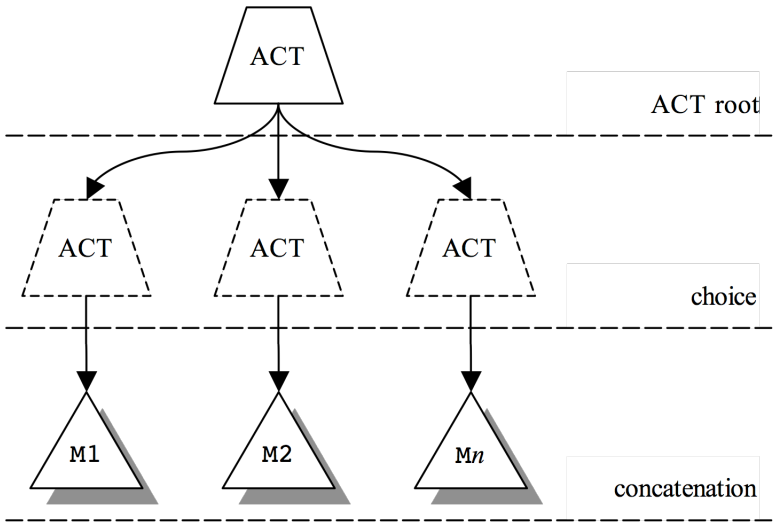

Figure 35: ACT with several possible messages

Here, there are *n* possible messages that could be sent by a particular role when the interaction is in a particular state. In order to allow the possibility for any of the messages to be composed, each concept added to the ACT must be passed to each of the alternate messages in parallel.

5.3.5. Literal Values Associated with Token Concepts

In the example presented in Section 5.3.3, concepts were added to the subtree of the ACT that represents `M`. Concepts are only half of the picture, however; without an associated literal value to accompany the concept, any token in the ACT with a matching concept will not necessarily be usable in the composition of the message, given that no literal value is necessarily available as a substitute for the token.

Consequently, when a concept is added to the ACT, it should be (in the general case) accompanied by a literal value. Figure 36 pictorially demonstrates the addition of this requirement (or pseudo-requirement, explained in the next subsection).

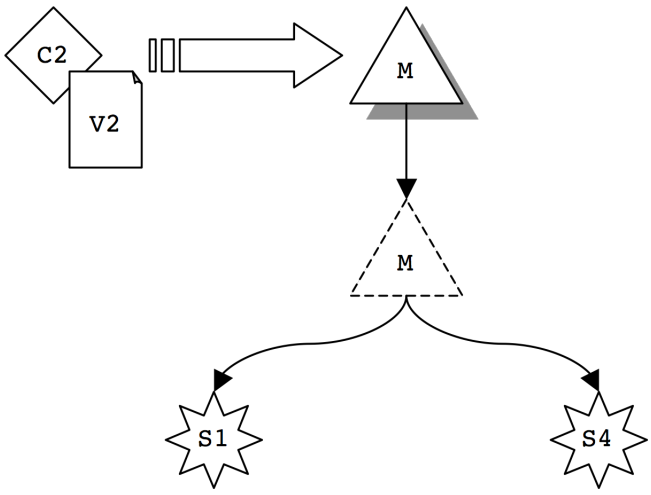

Figure 36: Adding a concept with an associated literal value

Of course, this introduces a possible inconsistency: A literal value provided with an added concept might not conform to the definition of the token production that is associated with a concept matching the added concept. In other words, a token that produces the literal value “X” might be associated with a concept that was matched by an added concept associated with the value “Y”.

Some solutions to handle such inconsistency present themselves; these will be dealt with in Section 5.3.10.6, where variants of the construction algorithm with respect to this issue are discussed. Alternatively, assuming some level of trust of the application that is providing concepts and corresponding literal values to the ACT, this “inconsistency” could potentially be used as a means of transformation, something that bears future investigation.

5.3.6. Default Literal Values Associated with Tokens

Presuming that an application is unaware of the precise structure of the protocol messages being sent, the likelihood that no message can be composed from

an ACT due to incomplete information is relatively high. Given this assumption, a mechanism by which missing information can be automatically supplied is not only a convenience but also a necessity; this mechanism is one of default (literal) values associated with the token productions of the ACT.

Default values serve multiple purposes. As mentioned, one such purpose is the further completion of an ACT in the circumstance that the ACT has been provided insufficient concepts (and associated values) in order to compose even one of the possibly several messages represented by the tree. There are actually three reasons for why an application might not supply sufficient information: The application may be ignorant of the concepts that would allow it to supply the missing information; the application may be ignorant that the ACT requires the missing information (i.e., the application does not have relevant information regarding the structure of messages represented by the ACT), even if the application is knowledgeable of the relevant concepts; or, the literal value that represents a "missing" token in a message derived from the ACT is identical under all circumstances, regardless of whether the application provides the corresponding concept (and thus it would be redundant for the application to add such a concept to the ACT).

In the first two cases, there is (presumably) potential variability in the literal values of the tokens, but one such value should be selected as the default if the token is required in the composition of a message. In the last case, the value of the token is a constant (i.e., the production that defines the token prescribes only one possibility) that can simply be used as the default. For example, the "HTTP" token in an HTTP request (or response) is invariable and yet must be included in every HTTP message;

it would seem redundant to require that such information be provided by an HTTP application. Most elements of "syntactic sugar" also fall into this last category.

A second purpose for default values is to allow the composition of a message involving tokens that are not related to concepts. No concept/value pair added to the ACT will ever be able to supply a literal composition value for such a token. In this case the *only* possibility is to associate a default value with the token, so that the default value can be used during composition. Note that it makes no sense for the production that defines such a token to allow for more than one possible string, since only one of the possibilities will ever be used. Further, this situation implies a very important condition for the purpose of composition: *Each token production must be associated with a concept, must have a determinable (via assignment, calculation, or otherwise) default value, or both*.

A third purpose for default values is as an alternative to supplying a literal value when adding a concept to an ACT. In the last section we described why, in the general case, a literal value should accompany a concept added to an ACT. This is instead the specialized case, when a literal value need not accompany a concept added to an ACT. If a concept is added to an ACT without a corresponding literal value, and a matching concept (to which some token in one or more subtrees of the ACT is associated) is found, the default value of each of the relevant tokens can be used instead.

The final purpose for default values is the forcible composition of a protocol message. Even with default values, it may be possible that no message can be composed from an ACT, and that at least one required token in each subtree of the

ACT that represents a message has no default value and is associated with a concept that was not matched by any concepts added to the ACT. In this case, the interaction can simply fail, or we can build into the protocol specification a message that would indicate such a situation or otherwise allow the interaction to continue normally. However, we likely do not want this message to be composed and sent in any situation other than the one described, so if *another* message can possibly be composed, then that one should instead be selected under all circumstances. The way in which this can be accomplished is by the specification of a key token of the aberrant message to use a default value only when a message must be forcibly composed in order to continue the interaction. Such a default value would not be used unless forced, and it would not be forced unless no message could be composed from the ACT.

The purposes and possible uses of default values suggest the following:

1) Each token, before prescribed only the two possible associations of a concept and (by the previous subsection) a literal value to be used during composition (when dealing with the ACT), is now also prescribed the third association of a default literal value. (Structures and messages still have only the possible association of a concept, though such is not relevant to the ACT.)
2) Each token cannot be lacking both the association of a concept and the association of a default value.

3) If a token is associated with a default value, the default value may be *constant* in the sense that it overrides any literal value provided with a matching concept.
4) If a token is associated with a default literal value, the default may be *optional* in the sense that it is not used to initialize the literal value associated with the token when the subtree of the ACT corresponding to the token is constructed.
5) If a token is associated with an optional default value, the default value may also be *forcible* in the sense that it is not used as the literal value associated with the token unless no matching concept with an associated literal value reached the token after having been added to the ACT, and a message must be forcibly composed.

To illustrate the association of default values with tokens in an ACT, Figure 37 shows the inclusion of default values for `T1`, `T2`, and `T4`, prior to the addition of concepts `C1` and `C5` (pictured in Figure 34).

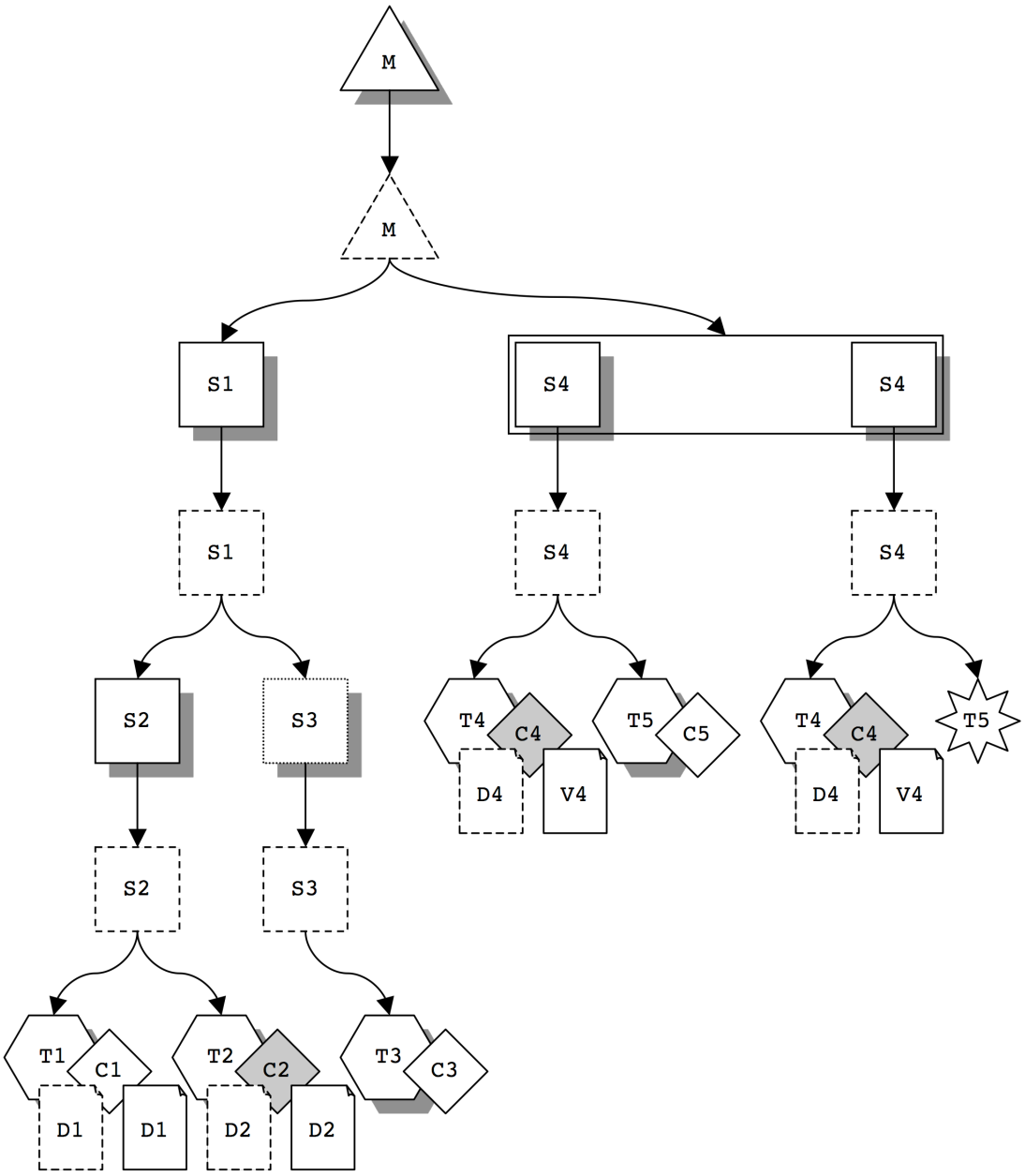

Figure 37: ACT with default literal values

This diagram depicts several additional things over the diagram of Figure 33. Perhaps the most obvious is that `T1` not only has an associated default literal value, but also an associated composition literal value, even though the concept with which `T1` is associated has yet been matched; this implies that the default literal value for

T1 is not considered optional, so the composition value of T1 was initialized with the default value, and no concept C1 (with or without an associated value) need be added to the ACT in order to include T1 in a composed message for M. The second thing to note is that the value "D2" associated with T2 is not only the default value but the composition value as well, despite the fact that the concept with which T2 is associated has been matched; this implies that either the default value associated with T2 is constant, or the matching token was added to the ACT without a corresponding literal value. Finally, we note that, for both instances of T4, the default value has been overridden by a value provided with a matching concept.

A curious situation arises when considering whether a constructed instance of a placeholder would have a default literal value and thus possibly an initialized literal (composition) value. With such consideration, a message *might* possibly be composed, yet without such consideration, a message *might not* possibly be composed. In our example, neither of the subtrees that correspond to the S4 principal rule nodes is complete, given that neither of the T5 nodes (at least one of which is required) has an associated literal value (the second instance not even having been constructed). In this case we can see that neither instance of S4 (paying particular attention to the second instance) can possibly be complete because T5 does not have an associated default literal value, and thus cannot possibly have an automatically initialized literal (composition) value. In the general case, we cannot know whether the defaults of a subtree to be constructed as a replacement for a placeholder would complete a section of the ACT without actually constructing the appropriate subtree.

5.3.7. Context-Sensitive Termination

Composition must handle our special case of repeated grammar element termination, in which one grammar element in an earlier part of the message indicates how many of another repeated grammar element occur later in the message. The mechanism by which this termination can be specified using the languages developed for this work was already presented in Section 5.2.4. The specification of such turns out to be trivial; the parsing of such is only slightly more difficult.

Composition presents a significant problem, however, given that we would like for as much automation of message composition to occur as possible. In other words, if *X* of some counted grammar element are included in an ACT, the grammar element (token) that keeps the count should automatically be set to *X* during composition. We can envision several situations that make this (supposedly) simple automation challenging, possibly even impractical.

To begin with the easiest issue, a subtree of the token that tracks the count may be optional. In this case, the intention of the specifier is probably that the token should only be excluded from the composition of a message when the count is some "normal" predetermined value, probably zero. However, the specification languages provide no clear mechanism to describe this behavior (though they could certainly be extended). Instead, the subtree in question may be complete without actually including the count, even when the counted grammar element is repeated fewer or more times than the intended (unspecified) default.

For the purposes of message composition as described here, the default count will always be considered zero. Further, message composition must ensure that, if the

count is not zero, the token is included in the composition even when contained within an otherwise complete subtree. Note that the count-tracking token must always be included, even when the count is zero, if no ancestor rule node in the message subtree of the ACT is optional.

A second, more difficult issue arises when the counter token is associated with a concept, and the literal value of the token has already been set by a matching concept added to the ACT. If the literal value of the token is the same as the count (in the appropriate format) of elements in the corresponding repeated sequence, no conflict arises. If, however, these two values are different, as shown in Figure 38, a potentially irreconcilable situation occurs.

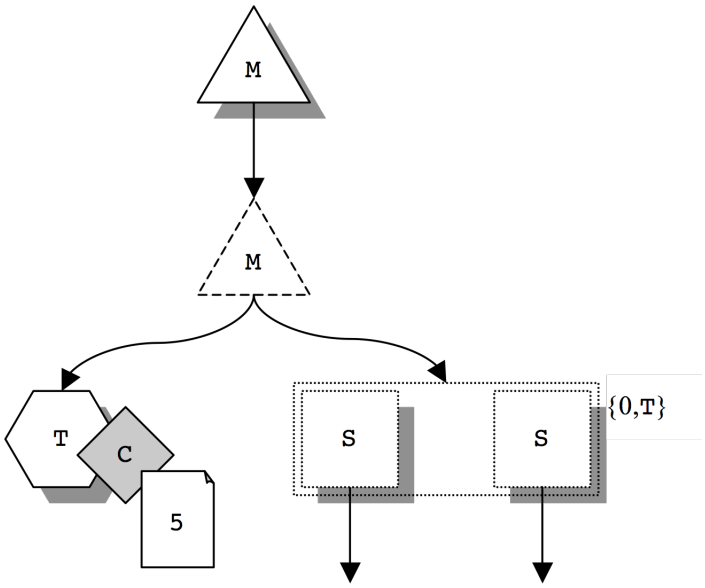

Figure 38: Conflicting count and repetition

According to the diagram, `S` is to be repeated anywhere between zero and `T` times, but the literal value of `T` has already been set by a match of `C`, to which `T` is associated. Even if both subtrees corresponding to a repetition of `S` are complete,

their count (of two) cannot compare to the set value of "5". Now the question is not regarding the completeness of the message subtree, but rather its consistency.

For our purposes, an inconsistent subtree of the ACT is treated identically to an incomplete subtree; that is, it prohibits the composition of the message that contains the subtree in question. However, the consistency presented above can be rectified by the addition of another node to the tree, as shown in Figure 39.

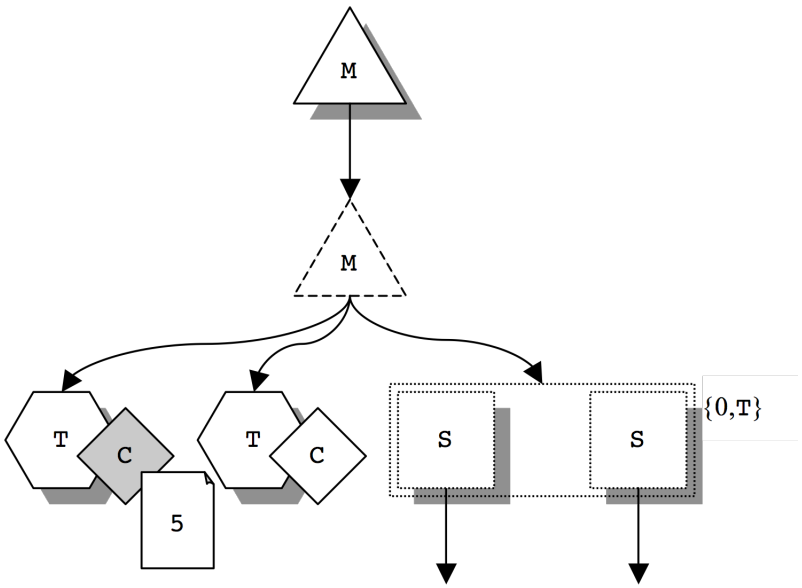

Figure 39: Possible resolution to conflicting count and repetition

Now, another instance of T exists (without an associated literal value), and the actual count can be associated with this instance. The inconsistency is resolved because the second instance of T (and thus the "correct" literal value) is the one closest to the repetition of the counted entity in a preorder traversal (or more specifically, parse order) of the completed subtrees.

The resolution introduces another potential problem, however: that of more than once instance of the counter token being included in the message, any of which

may be a preceding subtree to the repeated entities, and thus can be (partially) used as the specification for the repetition pattern of the rule in question. The choice of which one to use is further complicated by the fact that more than one sequence of repeated grammar elements can possibly use the same count. Such a situation is depicted by Figure 40.

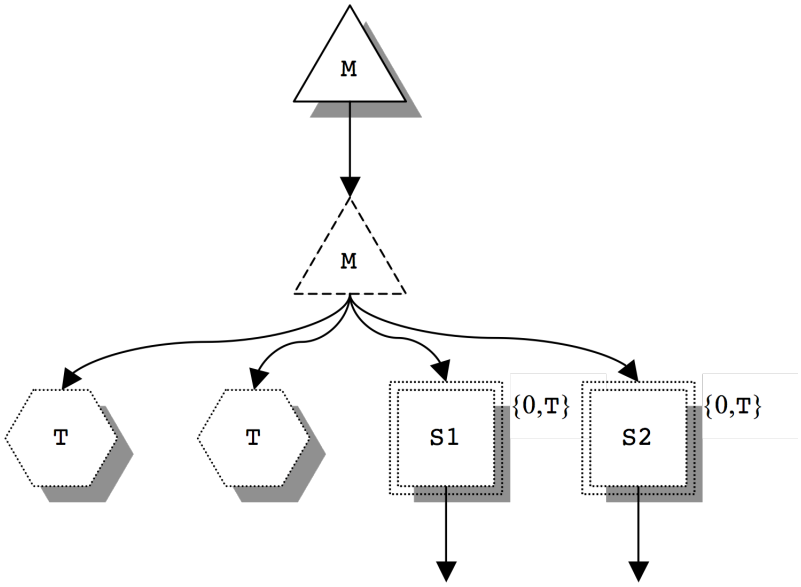

Figure 40: Multiple counted rules with the same count

The principal rules of both `S1` and `S2` have a repetition pattern defined by `T`, but more than one instance of `T` can possibly be included in a composition. If the composition includes different numbers of instances of `S1` and `S2`, we may be tempted to assign different literal values to the two instances of `T`, but this would cause a parsing conflict, as the last such instance overrides the count value of all previous instances. This situation can be rectified by using a different token (even if the definitions are identical) for each count, rather than `T` for both.

This problem is further complicated by the fact that such counted repetitions (and even the counts themselves, as subtokens) can be nested. Though we will not cover every possible conflict that can arise from the various combinations of nested and/or counted rules, multiply defined counts, etc., suffice it to say that, in the general case, composing a message that contains counted grammar elements is an unsolvable problem. Consequently, we currently require that counter tokens be associated with a concept and the count be explicitly given by the application (or a default value) in the form of an added (matching) concept, and that this count is consistent with the number of repeated grammar elements. Limitations and clarifications to the issue of context-sensitive termination will be given further study as part of future work.

5.3.8. Message Composition

Up until now, most of the composition issues discussed have actually been ones involving the construction of the ACT that will be used to compose a protocol message, rather than the process of composing the message itself. However, once the ACT is sufficiently constructed, a protocol message will likely need to be composed in order to continue with the interaction.

A protocol message is composed by concatenating token literal values at the lowest levels of the tree. This requires more than the generation of a mere unparser, commonly used for formatters or "pretty printers" [BV96]. Even without the fact that additional construction may take place during composition, an ACT need not be fully annotated with literal values (i.e., represent a valid syntax tree) in order to extract a protocol message; we require only that at least one principal rule node that corresponds to the definition of a message be "complete". A message rule node

(including the principal rule node) is complete if either of the following conditions holds (though the first actually implies the second):

1) The repetition pattern of the rule is optional (i.e., the repetition pattern specifies a lower bound of zero).
2) The number of complete children (message alternate nodes) is greater than or equal to the lower bound of the repetition pattern of the rule.

A message alternate node is complete if all of its children are complete. The children of a message alternate node may be a message rule node (though not a message principal rule node), a structure rule node, or a token rule node.

A structure rule node is complete if a condition analogous to that of the completion property for a message rule node is satisfied. Also, just as for a message alternate node, a structure alternate node is complete if all of its children are complete. The children of a structure alternate node may be a structure rule node or a token rule node.

A token principal rule node is complete if any of the following conditions hold:

1) The repetition pattern of the rule is optional (i.e., the repetition pattern specifies a lower bound of zero).
2) The number of complete children (message alternate nodes) is greater than or equal to the lower bound of the repetition pattern of the rule.
3) The node is annotated with a concept that has been matched by a concept added to the ACT. If the node is also annotated with a constant default value, the default value will be used in the composition. In the absence of

a constant default value, if the added concept was accompanied by a literal value, the provided literal value will be used; otherwise, the (non-constant) default value will be used. (In the case where an added concept, without a paired literal value, "matches" the concept associated with a token principal rule node for which no default value is available, the concept will not technically be matched, as no literal value can possibly be determined for the purpose of composition.)

4) The node does not have a determinable composition literal value as per the above rule but is annotated with a non-optional default value. In this case, the default value will be used in the composition.
5) The node does not have a determinable composition literal value as per the above rule but is annotated with a forcible, optional default value, and a message is being forcibly composed.

Note that the above criteria apply only to token principal rule nodes, since these nodes can be associated with a concept; token (sub-)rule nodes are complete if at least one of the first two criteria hold. The completion criterion for token alternate nodes is the same as that for message alternate nodes and for structure alternate nodes.

Note that if a principal rule node is represented in the ACT by a placeholder node, the ACT may still be complete, given that default values can "fill in" portions of the tree that have not been reached by (matching) concepts. Figure 41 shows a message subtree of an ACT that may (or may not) be complete, depending on the specifics of the second placeholder node (`S2`).

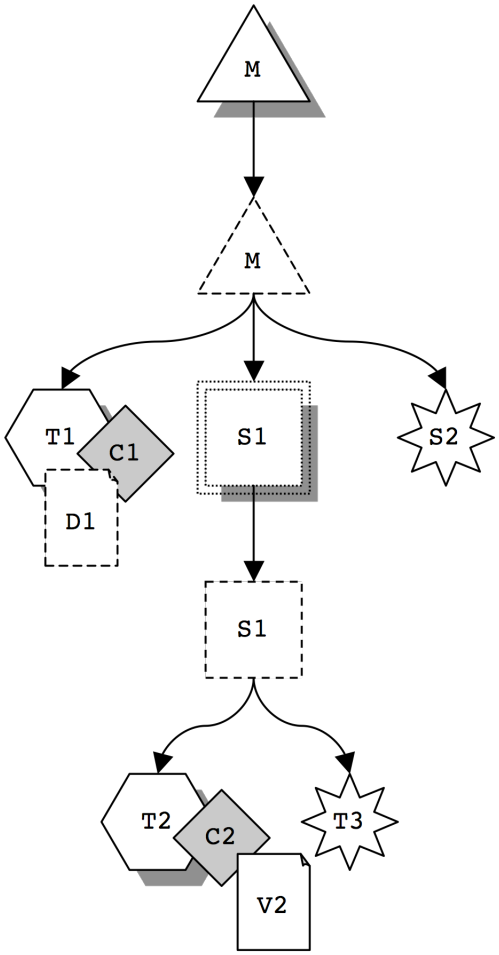

Figure 41: Ambiguously complete ACT

The first direct subtree of the message alternate node is complete, since the concept with which the corresponding token is related has been matched (and a default value is available). The second direct subtree is also complete, since the repetition pattern of `S1` is optional (making the unresolved placeholder node for token `T3` irrelevant). The third direct subtree *may* be complete, depending on the subtree instantiation of `S2`.

In the general case, an "empty" (non-recursive) construction of the ACT must take place during the composition of the message, in order to determine if the message is indeed complete.

5.3.9. Whitespace

As previously mentioned, whitespace is any token filtered by the lexer as irrelevant to the parser (regardless of whether the token is composed of "space"). While whitespace is irrelevant from the perspective of the parser, it is not irrelevant from the perspective of the composer.

Whitespace is used for two purposes: overall aesthetics and token separation. The first is of no importance for automatic composition, but the second is relevant. Suppose we have the following token and structure definitions:

```
T1 ("A" | "AB" | "ABC")
T2 ("CD" | "E")
T3 ("BC")

S1 (<T1> <T2>)
S2 (<T1> <T3>)
```

If two tokens follow one another in a larger structure and the leading portion of the second token can be confused as a continuation of the first token (such as with `T1` and `T2` in `S1`), an intervening token that acts as a separator ensures no ambiguity during lexing. For example, given the string "`ABCD`", a (longest-match) lexer would match "`ABC`" as `T1`, leaving "`D`" as an unrecognizable token. A whitespace token (irrelevant from the perspective of parsing) between "`B`" and "`C`" would disambiguate this situation.

If the entire second token can be confused as a continuation of the first token, a separator is necessary to prevent a possible parsing ambiguity. For example, given

the string "ABC", our lexer would again match "ABC" as T1, causing a parsing error when no further token was forthcoming. A whitespace token between "A" and "B" would disambiguate this situation.

Of course, not every two contiguous tokens require an intervening separator. The string "AE" is uniquely broken into the two tokens T1 ("A") and T2 ("E") with no backtracking.

So, in the absence of whitespace separators, two adjacent tokens can cause a lexing fault, a parsing fault, or no fault. The protocol developer must be aware of these possibilities, given that the expressivity of the languages does not prohibit such ambiguities from occurring within the specifications (though certain analyses can be performed [Cla98]). We will assume that the developer has identified whitespace separators that disambiguate any potential overlap between two adjacent tokens.

With regard to the ACT, sequence nodes can be annotated with a whitespace token, the default value of which is used as a separator between each pair of adjacent tokens present in the composition. Figure 42 shows an ACT message subtree with whitespace annotations.

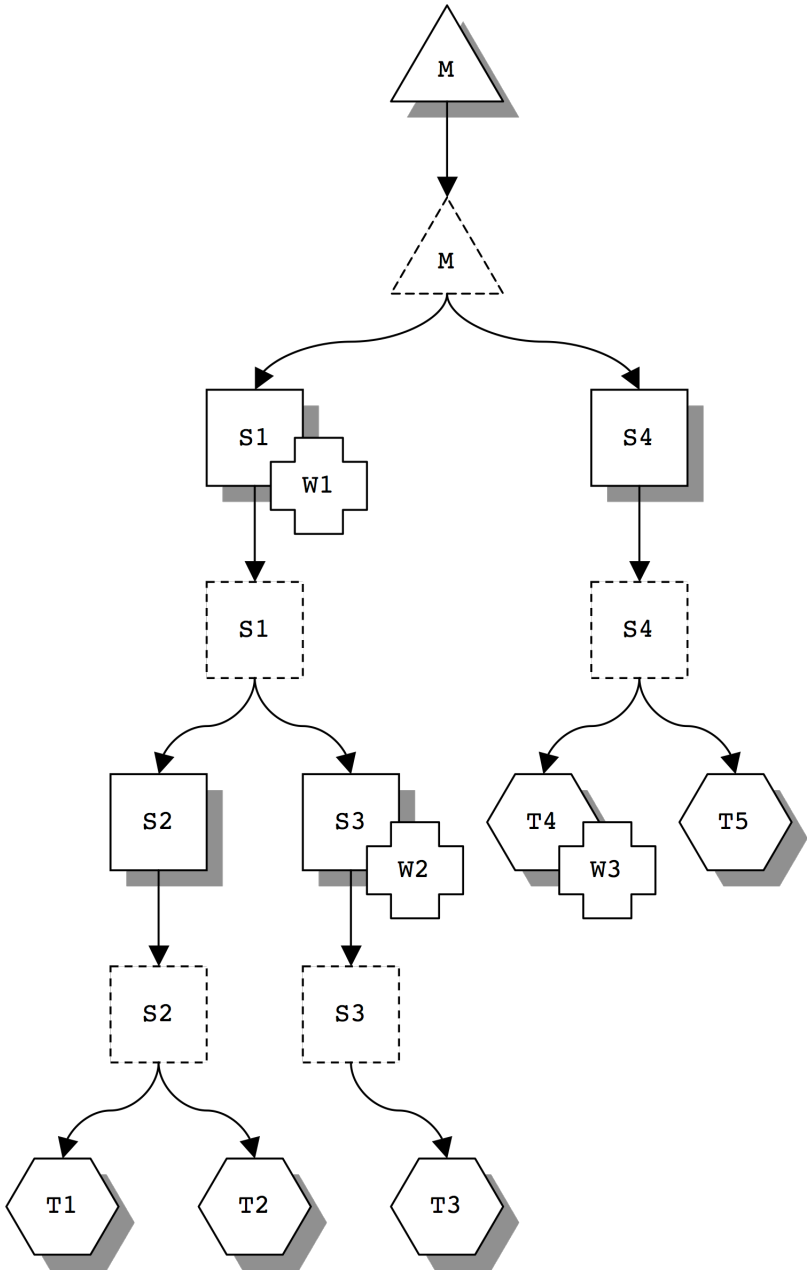

Figure 42: ACT with whitespace annotations

The default literal value associated with the whitespace token is appended to each token (of the subtree) included in the composition. Assuming the above subtree is complete, the composed message will follow the pattern

"`T1 W1 T2 W1 T3 W2 T4 W3 T5 W3`". Recall that of the three possibilities that resulted from an absence of whitespace separators, one was a token overlap ambiguity. However, unless we perform an analysis on each pair of literal values associated with two adjacent tokens to ensure that the leader of the second cannot be parsed as a trailer for the first, we cannot be certain that this third possibility does not hold in any given message. Since the lexical decomposition of a byte stream into tokens is a relatively fast procedure compared to parsing, and in order to save the time of pair-by-pair analysis of composition literal values, we will place a whitespace separator (if one is specified) between every pair of composition literal values, regardless of whether such a separator is necessary. (For example, assuming a whitespace separator of a single space, the ACT representing a string such as "`{A:A:A}`" would be composed as "`{ A : A : A }` ", even though the separators are technically superfluous according to the implied definitions of the tokens involved.) Note that if no whitespace token is specified in the definition of the message and nested structures (or if an empty whitespace token is specified), the literal values will be composed without any intervening separator.

5.3.10. Construction Algorithm Variants

There are several possible variants to both the construction algorithm and the composition algorithm. Some have legitimate use and others do not. The variants that provide us with useful or interesting alternatives will be explained in detail; others that might provide us with some semblance of boundaries to these processes may be introduced insofar as the reasoning behind their elimination can be explained.

This subsection will describe interesting important variants to the construction algorithm, whereas the next subsection will describe variants to the composition algorithm.

5.3.10.1. Alternate Traversals

When adding concepts to the ACT, we perform a preorder traversal of the tree (with the exception of the breadth-first "slice" of the tree when traversing recursive structures). This is consistent with the way in which productions are both written and parsed, but ultimately arbitrary; messages and subproductions need not be composed in such a way. The two other methods of traversal that immediately present themselves are post-order traversal and breadth-first traversal.

A post-order traversal has no obvious advantage or disadvantage over a preorder traversal, except perhaps that a preorder traversal is more intuitive. In fact, the difference between these two traversals is minimal, restricted only to timing issues (the more favorable possibility subject to the specification of the messages) if no two tokens of a message possibly composed by the ACT are associated with the same concept. (This property of unique concept attribution will become a theme when discussing many of the variants.) In cases where two (or more) tokens are associated with the same concept, the order of the traversal will determine which of the tokens will have its literal value set.

A breadth-first traversal actually has a disadvantage over a preorder traversal: The potential for completing a subtree is reduced, since concepts are necessarily spread across multiple subtrees before a single subtree is examined. Again, such a traversal has no impact on an ACT given unique concept attribution.

Of course, there are many other orders of traversal that one could imagine; we have presented only two. However, given unique concept attribution and non-recursively defined structures, the ACT constructed by one traversal order will be identical to that constructed by any other traversal order, though the efficiency with which the ACT is constructed may be different. Differences arise if and only if either more than one token is associated with the same concept or if one or more structures are recursively defined. In the first case, the traversal merely imposes a (possibly partial) ordering on the sequence in which these tokens are matched by the given concept; in most circumstances, our selection for this order is arbitrary (though we will cover a special case in the next subsection). In the second case, a breadth-first traversal (or some other definite-depth traversal) is necessary in order to prevent an infinitely recursive traversal of the structure. Note, however, that two or more recursive structures at the same depth must also have an ordering, which can be determined by, for example, a preorder traversal.

Finally, it is important to mention that our selection of a preorder traversal is arbitrary particularly with regard to the components of a rule. Every rule is a set of alternates in the technical sense of the word “set”. While the components of an alternate are ordered, the components of a rule are not, which means that every traversal order of the alternates of each rule is equally valid. To rectify this ambiguity, we specify a preorder traversal as operating on all grammar components, including those of rules, in declaration order.

5.3.10.2. Multiple Tokens Associated with the Same Concept

This possibility, more precisely a construction issue rather than a construction variant, was introduced in the last subsection. We mentioned that the order in which such tokens will be matched is determined by the traversal, though this is not always the case for such tokens *in the same subtree*.

Because tokens cannot be defined recursively (according to our specification languages), if two principal rule nodes of the ACT are associated with the same concept, the token of one must (partially) define the token of the other; the "descendent" token is a subtoken, and the ancestor token may be a subtoken or a top-level token. Figure 43 depicts this situation: `T1`, a top-level token, is defined in terms of the subtokens `T2` and `T3`; however, both `T1` and `T3` are associated with the same concept.

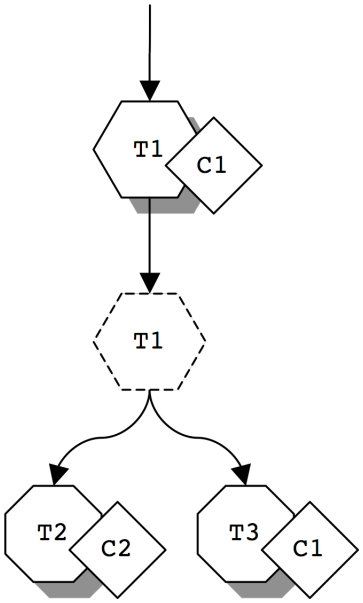

Figure 43: Token and subtoken associated with the same concept

Regardless, the ancestor token is matched first in a preorder traversal and the descendent token is matched first in a post-order traversal. While this in itself is not particularly significant, what is significant is the way in which we deal with principal rule nodes when the concept to which the corresponding token is associated has been matched by a concept added to the ACT. If, as specified, the traversal of a subtree of an ACT does not progress beyond a principal node that has a matched concept, then the matching of the ancestor token precludes the matching of the descendent token. So, adding two concepts to the ACT when using a preorder traversal, the ancestor token would be matched and the descendent token not (since the second traversal would not continue past the ancestor node), whereas using a post-order traversal, the descendent token would be matched and the ancestor token would be matched (in that order). In this way the traversal defines not only an order in which tokens can be matched, but even *whether* certain tokens can be matched.

This particular issue of tokens being "recursively" associated with concepts (even though a recursive grammar for tokens is disallowed) is expected to be unusual, however. Even the multiple occurrence of a token concept within a message subtree of an ACT is discouraged, particularly given that token position within a message can be considered part of the token's context, thus differentiating tokens that might otherwise represent identical concepts. Still, one possible variant of the algorithm would allow a traversal to match concepts beyond a node with a concept that has already been matched.

5.3.10.3. Overriding Literal Values

The last section reviewed the decision to not progress beyond a principle rule node if the node in question already has an assigned composition literal value. There is another, related variant that we can consider: A concept reaching (and matching) a node that has already been matched can override the old literal value (with the new literal value passed along with the subsequent concept, presumably), rather than either continuing with the traversal, either below the node or back to its parent.

This is an unrealistic possibility, however, if we consider that more than one token can be related to the same concept. If this variant were used, only the first such matching token, according to the traversal, would *ever* be set; all other instances would never be reached by the traversal.

5.3.10.4. Hints

One of the presumptions under which we have been operating is that an application is ignorant of or uninterested in the larger structure of an ACT, only being concerned with token concepts and the associated literal values. Instead, we could

also associate structure concepts and message concepts with the principal rule nodes of structures and messages, respectively, and allow an application to make use of such associations by providing "hints" for subtrees of the ACT to be either visited or pruned during the traversal.

An application adding concepts to the ACT could provide hints as structure concepts or message concepts, or perhaps a sequence of such concepts or even a set of sequences of such concepts. The concepts could then be used during the ACT traversal to identify which subtrees to either include or ignore, thus more specifically targeting certain nodes and avoiding some of the potential conflicts mentioned above. Such a situation is shown in Figure 44, where token `T3` is unambiguously associated with the literal value "`V`", even though every token (even those before the determined token in a strict preorder traversal) is associated with the same concept.

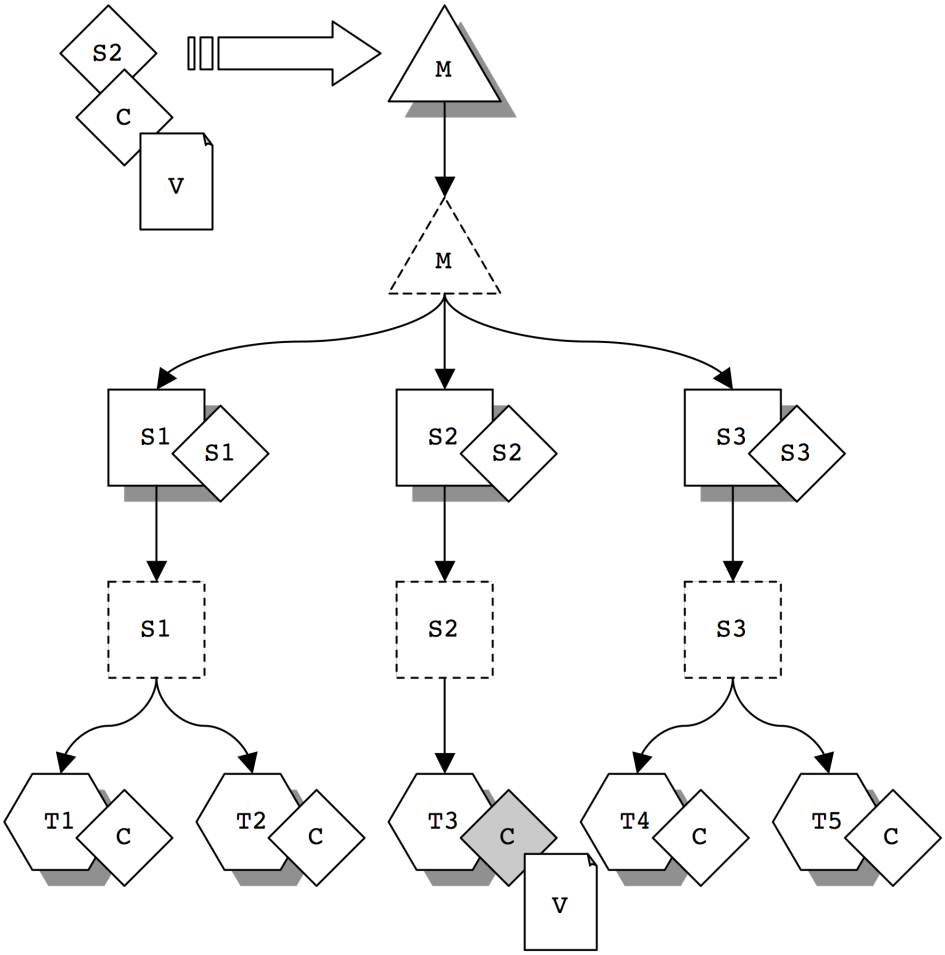

Figure 44: Use of structure concepts for composition

The implications of such a variant will be investigated as future work.

5.3.10.5. Parallel Construction

One of the difficulties inherent in the system described is the potentiality for constructing partially complete message trees that, if the traversal was different than the one specified, could have resulted in a single complete message tree. Consider the message subtree of an ACT pictured in Figure 45; note that both `T2` and `T4` are associated with the same concept, `C2`.

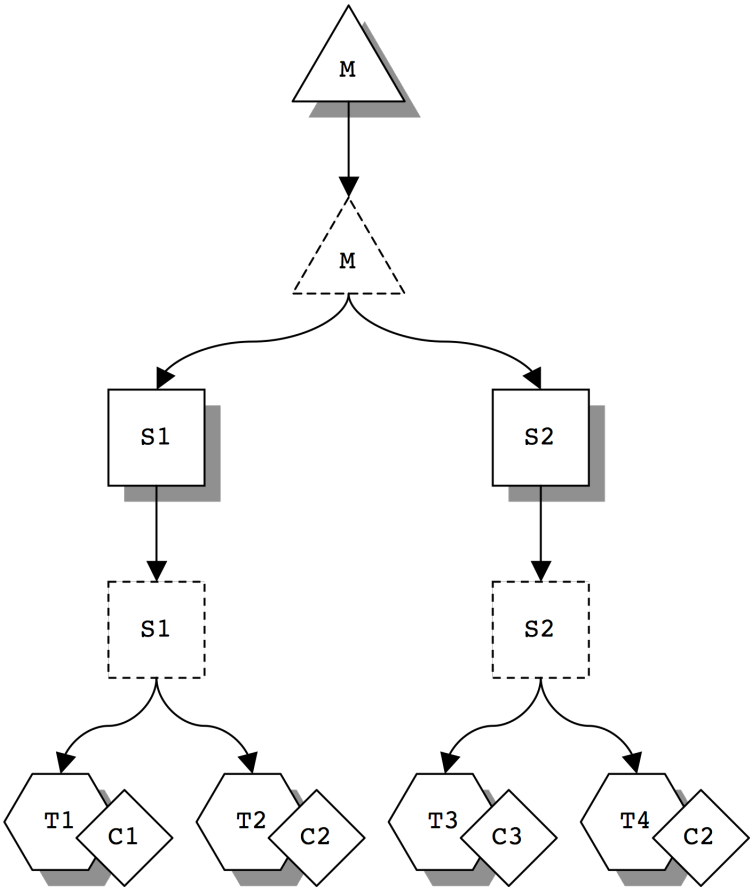

Figure 45: Potential for parallel construction

Were we to add the concepts `C3` and `C2` (with appropriate literal values) in that order (using a preorder traversal), `C3` would be associated with the first token of `S2`, and `C2` would be associated with the second token of `S1`; both of the structure subtrees are incomplete. If instead we altered the traversal being used to, say, a postorder traversal, the subtree corresponding to `S2` (and thus the entire message) would be complete.

Rather than rely on the somewhat arbitrary selection of traversal, we could instead allow each concept to traverse the entire tree, "setting" the literal value of *every* matching token, rather than only the first one found by the traversal. In this case

a tally would be kept of how many times each concept had been added, and certain efforts could be made to maximize the completed subtrees. (An optimal solution is, in the general case, computationally infeasible, however, so heuristics would need to be used.)

This variant is another feature considered for future work.

5.3.10.6. Literal Validation

One condition that is assumed by our algorithm is the consistency between the literal value passed with a concept added to the ACT and a possible string produced by the token that is associated with a matching concept. This condition is assumed for the purpose of efficiency; otherwise, every literal value added to an ACT would need to be lexically analyzed at the point of every potential match.

Instead, we can require such lexical analysis (which will also be performed during the parsing of the composed message, only at the target end) as an algorithmic variant. This will, in effect, at least double the amount of lexical analysis performed by the system; however, it will also ensure that each literal value used to compose a message is consistent with the definition of the corresponding token. This is a “safe” approach to message composition, particularly in light of the fact that we place no restriction on the application to necessarily have knowledge of syntactic elements, including tokens. Though we prescribe no typing mechanism for concepts, the semantic properties assumed by a concept with which a token is associated should imply the appropriate format (the definition of said token being presumably consistent with the implied format of the concept, of course), and we expect that an

application understanding of such a concept is also understanding of the implied format.

5.3.10.7. Variant Combinations

Many of these variants can be combined successfully. While the potential combinations are exponential in terms of the number of possible variants (only the most important of which have been mentioned), certain variants are particularly compatible with one another (e.g., alternate traversals and hints), whereas others (such as traversing beyond a matched concept and parallel alternate composition) may provide some interesting difficulties.

Regardless, it is worth mentioning that none of these variants (except for those that are impractical, such as overriding a literal value by multiply matching a concept) necessarily precludes any of the others.

5.3.11. Composition Algorithm Variants

Just as variants can be conceived for the construction algorithm, so the potential for variation exists within the composition algorithm. Some of the more important variants are discussed below.

5.3.11.1. Subtoken Composition

One of the issues mentioned in the discussion on some of the possible variants to the construction algorithm was the potential for subtokens to be associated with the same concept as a top-level token (or another subtoken) that references the subtokens in its definition, and how the traversal order determines whether subtokens are matched. Even without the condition of nested concept associations, any traversal order might match the concepts of subtokens before matching a concept of the

ancestor of those subtokens (if the ancestor has an associated concept, of course). Figure 46 shows such a scenario, including the literal values of the principal rule nodes that were set when the concepts of each node were matched.

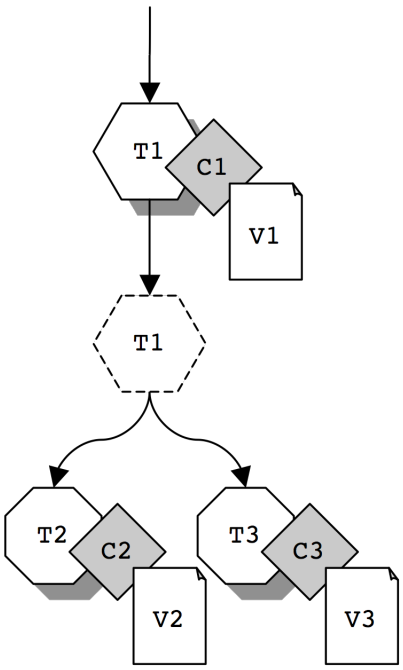

Figure 46: Multiple composition possibilities

The subtree shown is doubly complete: The matches of `C2` and `C3` (combined) complete it in one sense (composing the string "`V2V3`"), and the match of `C1` completes it in another sense (composing the string "`V1`"), albeit at a higher level in the tree. The given composition algorithm specifies that the highest level of completion should be used; however, an algorithmic variant could allow the lowest level (or any other level) to be used.

5.3.11.2. Message Selection

It is possible that more than one message subtree of the ACT is complete. The choice of which message to compose and to send (given that the sending of any of the

relevant messages maintains valid interaction state) is arbitrary. Our algorithm prescribes that the first (according to declaration order) complete message be composed, though any is legitimate; consequently, a variant composition algorithm is free to select any such completed message. (A similar situation occurs with alternates of a rule, when the rule requires *m* of *n* completed alternates in the composition, $m < n$.)

5.3.11.3. Maximal Versus Minimal Messages

Optional and repeated rules allow for flexibility with respect to the composition of a message. If the subtree corresponding to an optional rule is complete, it can (by definition) be included or excluded in the composition of the message. Repeated rules can include as few or as many completed alternate subtrees as its boundaries permit.

Our algorithm specifies that, given the available completed subtrees, the largest message possible be composed (with one important exception). A variant of the composition algorithm might skip optional subtrees and include only those completed alternate subtrees of a repeated rule such that the lower bound of the rule was satisfied. (Of course, this is likely to exclude information that an application intended to be part of the message.)

The exception to the "maximal" algorithm involves subtrees that can be considered complete even without any concepts being added to them. This situation, shown in Figure 47, occurs when non-optional default values are associated with every token node necessary to complete the subtree.

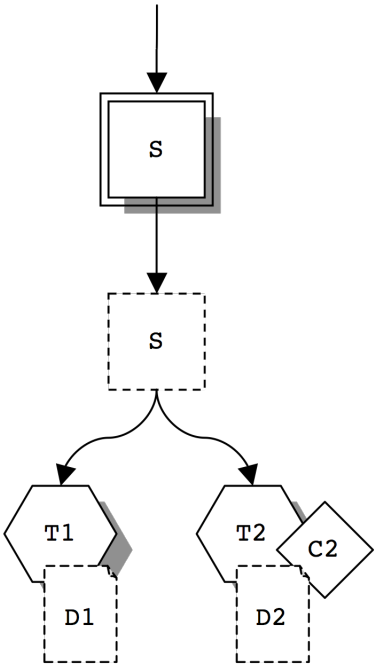

Figure 47: Initially complete ACT subtree

Neither of the token nodes has been matched by concepts added to the ACT (in fact, it is not even possible to match the first one, given that it is not associated with a concept), yet the default literal values, specified as non-optional, can be used as the composition value, thus allowing the subtree associated with S to be composed (as "`D1D2`"). The repeatability of `S`, also shown in the figure, means that one or more of these subtrees should be composed. (A similar situation would occur if `S` were specified recursively.) Unfortunately, without matched concepts to indicate how many repetitions should be included in the repetition, we could compose this subtree an infinite number of times, given that each instantiation of the repetition is initially complete. In such situations, our composition algorithm specifies that optional rules should be included, repeated rules should be included a number of times equal to their lower bound, and recursively-defined structures should be completed to a minimum

depth. A variant algorithm, on the other hand, can specify any number of inclusions within the boundaries of the repetition pattern. In practice, however, such constructs should be avoided, as they are unnecessary. (For example, the default associated with `T2` in the figure could be specified as optional, thus making the subtree incomplete unless a concept matching `C2` was added to the ACT.)

5.3.11.4. Partial Composition

One can recognize obvious similarities between an ACT and syntax-tree approaches to composition, such as, for example, the Document Object Model of XML (though a DOM tree is specific to the XML language, rather than to the language described by a DTD or Schema [Woo99]). A completed tree (or subtree) is not only used to compose the message, it is even used to determine which message is composed. This means that the message to be sent by a particular role during any given state of an interaction cannot be determined until the entire message is actually composed.

However, this approach precludes an optimization that allows a message to be sent even as it is being composed, something not uncommon. For example, HTTP servers configured to handle (large) dynamic content typically do not construct an entire message before the beginning of the response is sent; in fact, certain features of HTTP 1.1 were added specifically to handle this (e.g., the “chunked” transfer encoding [FGM+99]). In order to permit such an optimization, two issues arise.

The first issue is whether it is possible to predetermine the message that will be sent (assuming more than one message can possibly be sent) prior to the subtree of the ACT that corresponds to the message being complete. Certainly special

information can be received from the application that more specifically identifies the intended message. This information, whether inclusive (prune subtrees without “X”) or exclusive (prune subtrees with “X”), can be used to reduce the size of the ACT until only a single subtree corresponding to a single message remains, thus identifying the message that will be sent (presuming that the message subtree will eventually be completed).

The second issue is whether parts of the message subtree can be completed and considered finalized before other parts that will be used to compose a “later” portion of the message. To illustrate, Figure 48 shows an incomplete message subtree.

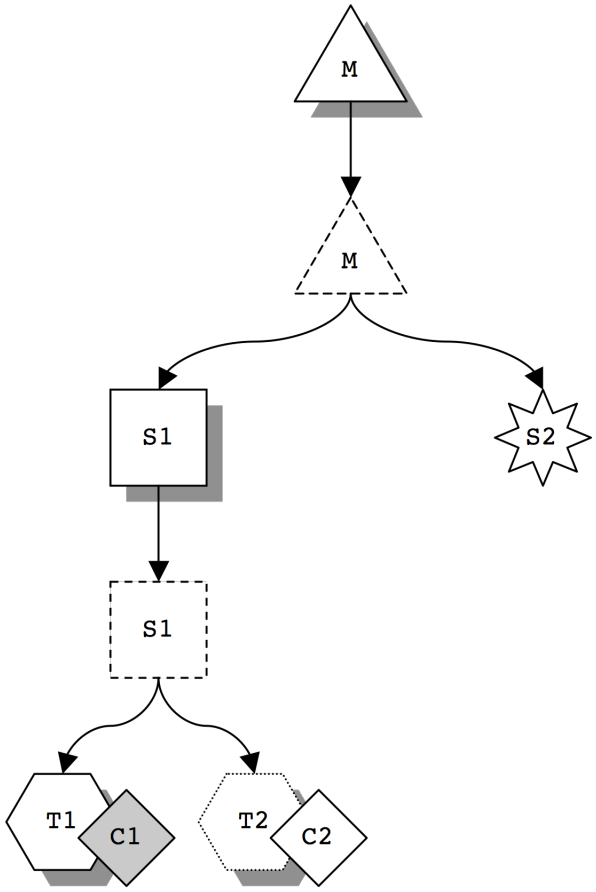

Figure 48: Left-complete ACT subtree

The subtree corresponding to `S1` can be considered complete, since `T1` has been matched and `T2` is optional. The subtree corresponding to `S2` may or may not be complete, but without instantiating the production as a subtree, we remain uncertain; thus, the subtree corresponding to `M` cannot be considered complete. However, at this point we could compose the first "part" of `M`, that part defined by the completed subtree of `S1`. Further, the partial composition could be sent as the beginning of `M`, provided no concepts added to the ACT would find their way to matching tokens in the subtrees that have already been composed and sent. (Prior to a

composed portion of a message being sent, such a match is permissible, given that the composition can always be recalculated; the sending is a point of no return, however.)

This variant of composition has downsides, however. The first is that knowledge of the form of messages (and substructures) is attributed to the application, something that violates past assumptions. The second is that, once a portion of a message has been sent, we require that the remaining "later" subtrees of the relevant portion of the ACT be eventually completed; if not, the message on the wire will remain incomplete, thus invalidating the interaction (whereas the possibility of a message that had previously been pruned from the ACT, if used, could have allowed the interaction to continue or terminate normally).

5.3.11.5. Variant Combinations

As with variants of the construction algorithm, variants of the composition algorithm can be combined successfully, particularly given that many of the choices made are, in essence, arbitrary. Of the variants presented here, the one that possibly offers the greatest potential for conflict is that of partial composition, particularly with respect to other (unmentioned) variants that deal with the mechanism of context-sensitive termination.

5.4. Translation

Translation is the overall activity of protocol recognition and conversion for the use of the protocol by an application, involving the coordination among the individual activities of interaction-state tracking, message parsing, and message composing. Consequently, any component (or set of components) that performs the task of translation for a given protocol must be capable of parsing messages,

composing messages, and maintaining (possibly across several connections) the state of interactions in which a collaborating role is participating.

We refer to such an implementing component (or set of components) simply as a translator. Each translator is capable of tracking the state of each interaction of the protocol from the perspective of each role that uses the protocol, in addition to being able to parse and compose each message of the protocol. In most circumstances, this all-encompassing behavior will likely be more than that required by any instance of a translator. We could create translators with more specific behavior as required, though the more general translator allows for simpler generation and ensures uniformity; the implications of tailoring the behavior of a translator to one or more roles will be left for future work.

Much of what needs to be said regarding translation has already been discussed in relation to interaction-state tracking, particularly with regard to whether the interaction requires (or is capable of dealing with) an outgoing or an incoming message. In addition to the requirement or possible choice of composing or parsing a message, there is at least one important consideration: determining the role for which the first message communicated over a new connection is intended. Consider the interaction shown in Figure 49.

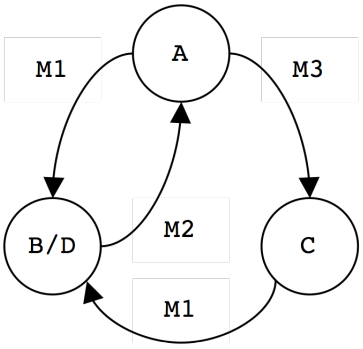

Figure 49: Potentially ambiguous interaction

First, notice that a single component implements both role `B` and role `D`. Also, notice that both the component that implements role `A` and the component that implements role `C` send a message `M1` to `B/D`. Given that the communication order is not intuitive, and given the further ambiguity in the diagram (introduced by the dual implementing component), the DFA, from the perspective of each of the roles, is shown in Figure 50.

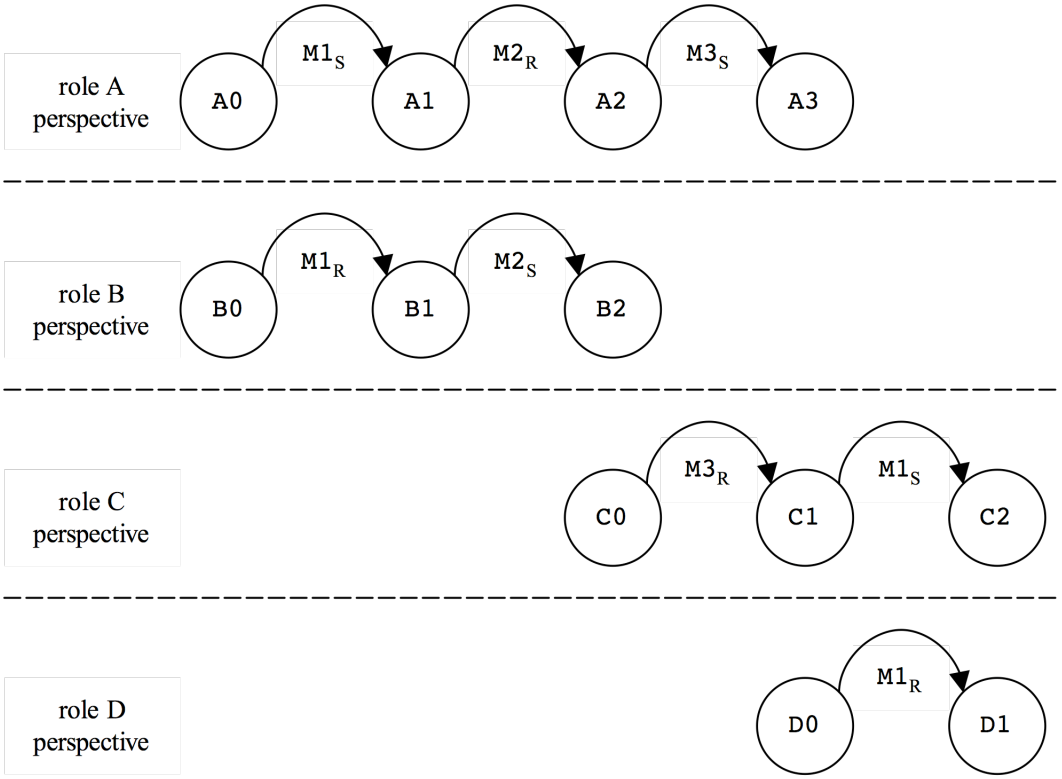

Figure 50: Potentially ambiguous interaction DFA

We can see from the diagram that the first instance of `M1` is sent from `A` to `B`, and the second instance of `M1` is sent from `C` to `D` (in that order, with intervening messages, of course). Thus, the component that implements roles `B` and `D` will receive both instances of `M1`. How, then, does the `B/D` component interpret the first instance of `M1` as coming from `A` and the second instance of `M1` as coming from `C`, given that each instance of `M1` is (presumably) the first message communicated over a new connection?

One possibility that immediately presents itself is simply to front-end each role with a different translator for the protocol. This is a legitimate method (although

possibly more resource intensive than desired) provided each translator separately maintains the interaction state for the respective role.

It seems unnecessary to multiply instantiate at a locale the same translator for a protocol, however, especially given the fact that each translator is capable of handling the interaction-state tracking for each role that uses the protocol, in addition to the parsing and composing of each message of the protocol. We prefer that one translator be able to handle protocol translation for both roles `B` and `D`. Consequently, we must provide a mechanism by which a single translator can differentiate between the two instances of `M1`, even though each arrives as the first message of a new connection.

Recall that communication between translators is purely messages of the protocol; no additional information, "meta" or otherwise, is permitted. As a result, we cannot front either instance of `M1` with information that would reveal the intended target role. We are then left with two possibilities for differentiating the messages: information within the message (i.e., the use of an interaction label) or the inherent properties of the connection.

While an interaction label might allow us to differentiate this case, it is only by implication. The first instance of `M1` would be communicated (from `A`) with an interaction label; if no interaction with such a label was currently ongoing, the message would clearly be intended for `B`. When the second instance of `M1` is communicated (from `C`) with the same label, an interaction with the label would be identified as in-progress, implying that the message is intended for `D`. While this method is capable of tentatively resolving the presented scenario, it breaks down if

more than two of the same message may be sent to the same component (which possibly implements several roles).

Thus, we are left with the connection itself providing the information necessary to determine the intended role. Although we cannot add meta-information to the messages communicated between translators, and we cannot guarantee that information within the message itself will determine the target role, we *can* require that the connection be established at a particular address (e.g., the role of an HTTP server listens at port 80), and consequently each potential target role can be assigned its own address. In this way, the simple act of connecting, in order to transmit the first (and subsequent) messages, uniquely identifies the target role.

It does not necessarily identify the source role, however. Consider the interaction shown in Figure 51.

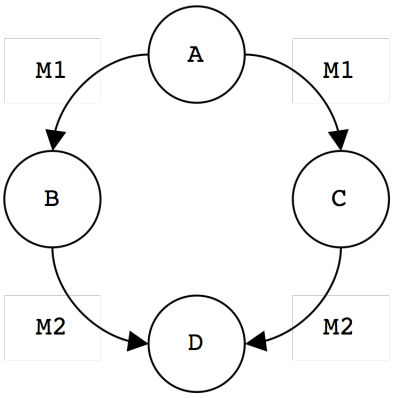

Figure 51: Ambiguous interaction

In this interaction, role `A` sends a message `M1` to each of role `B` and role `C`, and the two receiving roles in turn send a message `M2` to role `D`. The DFA for each role is shown in Figure 52.

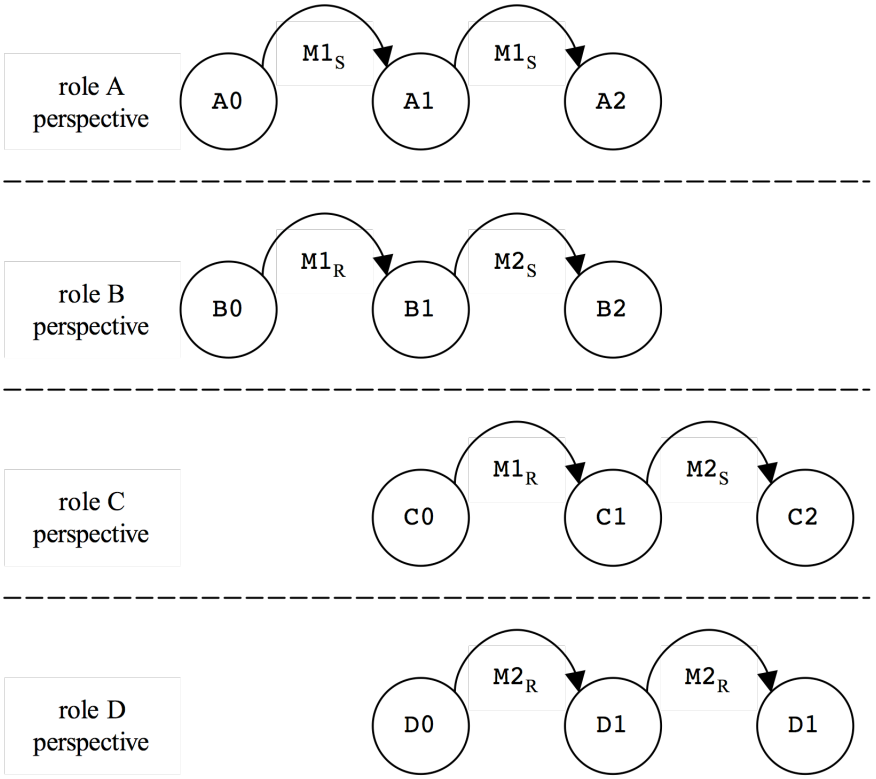

Figure 52: Ambiguous interaction DFA

While each of the role perspectives is deterministic, this is because the transitions are labeled only with the message being sent or received, and not also with the role to which the message is sent or from which the message is received. In this scenario, `D` may actually receive the messages in either order. If the source of either message is relevant, the DFA (from the perspective of `D`) becomes nondeterministic.

Thus, proper translation, specifically with respect to interaction-state tracking, relies on the source role of a message being either implicit or irrelevant. The target role, on the other hand, is necessarily implied by the translator address used to establish the connection by which the message is communicated. More on the translator will be said in the next subsection.

5.5. Generation

As mentioned in the previous subsection, a translator is capable of tracking the state of each interaction of the protocol (from the perspective of each role that uses the protocol), parsing each message of the protocol, and composing each message of the protocol. In this section we will discuss some of the issues with generating such a translator from a protocol specification.

Conceptually, a translator is a collection of components that perform the translation. These components are naturally suggested by the required functionality: a data structure that maintains state for each (current) interaction, a parser, and a composer. A translator must also include a means to interface with another (matching) translator, applications that implement the roles of the relevant protocol (in addition to somehow recording the means by which these applications are interfaced), and a negotiator (discussed in the next subsection). This is pictorially represented by Figure 53.

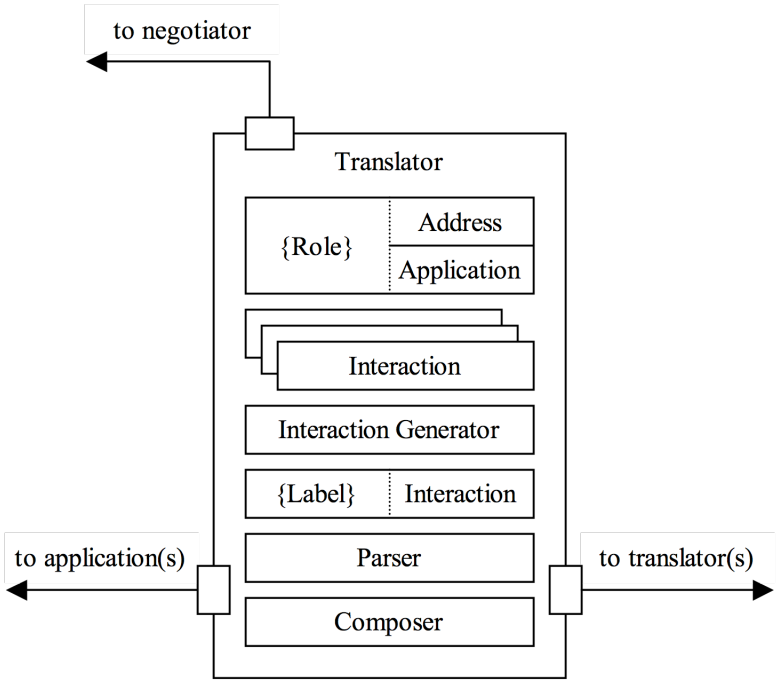

Figure 53: Translator anatomy

The data structure shown at the top is a conceptual map of the various roles implemented by applications for which the translator is acting as interpreter. Each role of the interaction is potentially associated with two pieces of information: the address at which the translator is listening for incoming connections from other translators, and the application component that implements the role. The generation of this data structure is trivial, though of course the generated translator must be capable of activating the listener addresses (either at the time of its generation or on demand) and connecting to the appropriate application components.

The interaction definitions and the interaction generator are shown below the role map; they are used to generate the components that represent (the state of) current interactions, which are associated with appropriate labels in the data structure shown immediately below the generator. (If the messages of an interaction are not

annotated with an interaction label, the interaction cannot take place over multiple connections, since there is no way to associate messages of the different connections as belonging to the same interaction; in this case the duration of the connection defines which messages belong to the same interaction.) Since a translator is specific to a protocol (which is defined by a limited set of interactions) the translator-generated interaction components are general in the sense that they can represent any of the possible interactions in any state from the perspective of any of the possible roles, but specific in the sense that they are tailored to the protocol. Thus, they can be defined as tailored classes in the OO design of an implementing system, in which case the "generation" of interaction-state tracking components would amount to the instantiation of the generated interaction classes. The generation of such classes follows directly from the interactions of the protocol as defined by the interaction specification language.

Of course, the generation of the interaction-state data structure is relatively simplistic; the data structure needs merely to be another map, this one of interaction labels to a component that represents the interaction in question. As explained above, the translator itself must be capable of generating these interaction components, by class instantiation or other means.

The parser is shown below the interaction label map. The generation of the parser follows directly from the specification of protocol messages using the message, structure, and token grammar specification languages. Compiler compilers are well studied and understood, so the details of generating a parser from the specifications

will not be covered in detail, though it is worthwhile to mention some validation aspects of parser generation.

The syntax of messages and structures is restricted to LL(1), and the syntax of tokens is restricted to that describable by a regular language. Given that the theory of compiler compilers is well established, we can use the corresponding principles and techniques to determine whether the message, structure, and token grammars defined by our specification languages are properly limited (e.g., the generation of the parser component of a translator is a good place to detect, say, left-recursive structures, which may define one or more grammar elements as more expressive than permitted). It is not necessarily the case that such validation needs to be performed at this stage. The evaluation of expressivity can be performed at any point during translator generation, particularly given that nothing prescribes the order in which the components are generated; the generation of the parser component is merely a logical point, given the known formality of the procedure.

At some point during generation, other validations that apply to both parsing and composing must be performed, such as the condition that a counting token appear before the structure whose context-sensitive termination is determined by the token. Otherwise, all of the parser-specific conditions discussed in Section 4.3 must be enforced at the time of parser generation.

As for the composer, shown at the bottom of Figure 53, several other conditions (discussed in Section 4.4) must be enforced at the time of its generation. Regardless, there are significant differences in the way in which the parser and the composer should be (independently) generated, despite their inverse function.

Whereas the parser deconstructs a message as a stream, the composer constructs a message piecemeal; placeholder nodes of the ACT are expanded during both the tree construction and message composition phases. Each of these placeholders represents a production. In order to expand the ACT, the placeholder nodes must be replaced by the subtree instantiations of these productions. This requires the composer to have a definition of each production available, and, of course, these definitions must be generated from the protocol specification. Furthermore, several roles may be attempting to compose a message simultaneously using the same translator, which, in conjunction with a component capable of generating an ACT, requires the translator to store a possibly partially completed ACT for each role. These additions are shown in Figure 54.

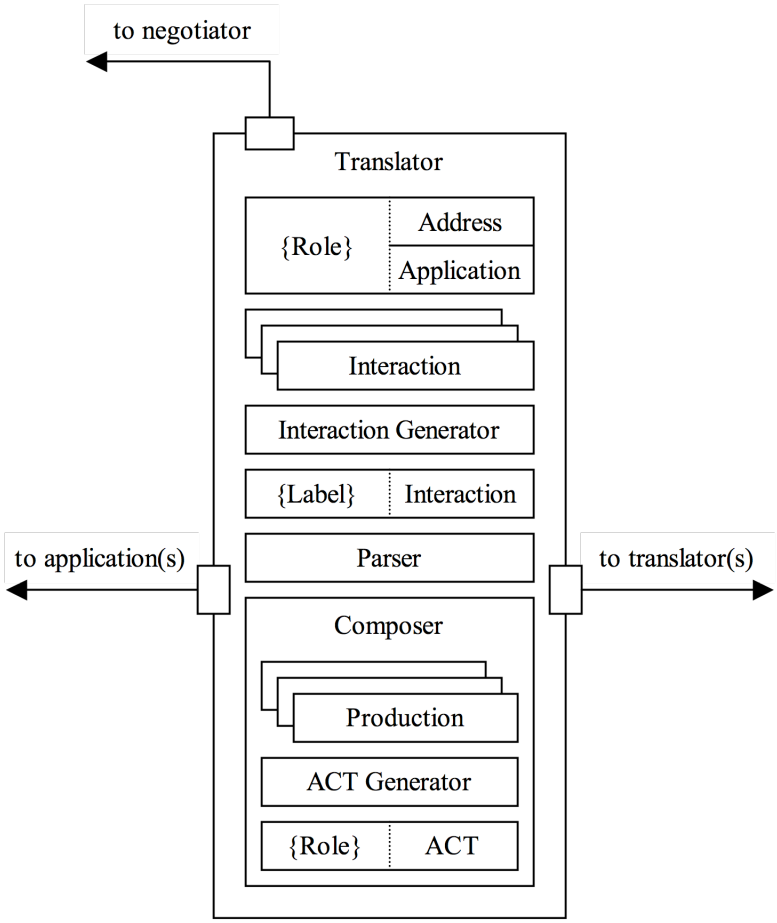

Figure 54: Expanded translator anatomy

The remaining aspects of the generated translator shown in the figure are the portals that allow the translator to interface with other translators, applications, and a negotiator. The translator must be generated with such a portal open to a negotiator, so that the negotiator can retrieve dynamic information from the translator. (The other portals can be created on demand during translator operation.)

5.6. Negotiation

Negotiation is the process by which one or more applications agree upon the protocol that will be used during an interaction. The process of negotiation is not the

focus of this work, but, given that two components must somehow determine which protocol to use in case of conflict, it was briefly introduced in Chapter 3.

We note again, however, that negotiation is not always necessary or even desirable. Given that a translator uses the actual protocol for remote communication, a translator should be able to connect directly to an application unaware of the concept-based approach (rather than with another translator). The three possible scenarios introduced by this interchangeability of application and translator (from the perspective of translator connectivity) were shown in Figure 18 on page 50.

We reiterate that in those situations conforming to the first case of the figure, negotiation may be performed if the applications prefer to use versions of the protocol that are different from one another. However, once two roles have negotiated a protocol, third and subsequent roles (which may not even be included in an interaction, depending) are then forced to use the negotiated protocol for the interaction in progress. Although it is certainly possible to include all potentially participating parties in the negotiation prior to the first message being sent, doing so before each role is included in the interaction not only may cause a communication burden, it may be wholly unnecessary. Still, we acknowledge this as a possibility for more complex protocol negotiations, as might parallel the negotiation exchange of the various dialects of SMB [Her03].

As with most choices, the arrangements of Figure 18 imply tradeoffs. The third case, which uses no negotiation and does not even require either application to be compliant with the concept-based approach, does not, of course, require the overhead of negotiation at the beginning of each interaction; it does, however, fix the

versions of the protocols in use, possibly requires an intervening (presumably hand-built) filter, and furthermore requires that each message sent from one application to the other be composed and parsed twice (composed by the sending application, parsed by the first translator, composed by the second translator, and parsed by the receiving application). The second case, which also uses no negotiation since one application requires a particular version of the protocol, has the advantage of eliminating the filter at the cost of requiring one application to be conformant to the concept-based approach and appropriately interfacing with a translator. The first case, on the other hand, allows for negotiation at the additional expense of requiring both applications to be conformant to the concept-based approach; the benefits are several, however, ranging from the dynamic addressing of the translators (which must have fixed addresses in the other three cases, presumably associated with well-known ports in order for remote applications to locate them) to a choice of optimization with respect to the negotiated protocol.

6. Prototype

In this chapter we will present the prototype developed to test the feasibility of this work. The first section will discuss the operation of the prototype using an implemented scenario in order to give the reader some context for the remainder of the chapter. The second section will briefly introduce the languages and tools used to construct the prototype. Remaining sections are devoted to the architecture of the prototype and the processes by which it operates.

6.1. Scenario

In this section, we illustrate the functioning of the prototype via an example application of our approach to dynamic protocol evolution. The scenario, actually implemented using the prototype event-based translation system, involves an encounter between an HTTP 1.1 client and an HTTP 1.0 server, the goal being to show how the client and server can communicate using their respective versions of the protocol via our prototype. (For details of the HTTP 1.0 and HTTP 1.1 protocols used for this example, see [BFF96] and [FGM+99], respectively.) The case study of HTTP will be further examined in Section 7.1.

Normally, an HTTP server that understands only HTTP 1.0 will have difficulty handling an HTTP 1.1 request. This is because servers are currently

designed to manage such a request in a way that is heavily tied to the syntax of the protocol. Thus, despite the fact that the two versions of HTTP are remarkably similar, even small changes in syntax have a serious impact on compatibility. On the possibility that the syntax would be misunderstood, HTTP 1.0 servers typically were programmed to deny all requests that did not specify exactly an HTTP 1.0 version number, irrespective of the later clarification of HTTP version numbers [MFG+97]. In theory, however, many (if not most) HTTP 1.1 requests could be accommodated by an HTTP 1.0 server, since the syntactic differences are minor and are almost exclusively additions to HTTP rather than modifications or deprecations.

Our example assumes that the client and server are in the first configuration described in Section 3.3. This means that the client and server are both written to operate on protocol concept events rather than on raw HTTP messages. The client communicates through an event-based translator using HTTP 1.1 concepts, while the server communicates, initially, through a translator using HTTP 1.0 concepts. In our implementation of the example, the client and server are simulated, in the sense that they each only communicate with their corresponding translator via a set of events limited to the minimum required for the most basic HTTP messages, performing no real HTTP-like processing, such as returning a page of HTML text in response to a GET request. The translators are generated from specifications of HTTP 1.1 and HTTP 1.0. These specifications, which are available in Appendix D, use the description languages discussed in Chapter 4 and defined in Appendix B.

The essence of the scenario is depicted and numbered in Figure 55.

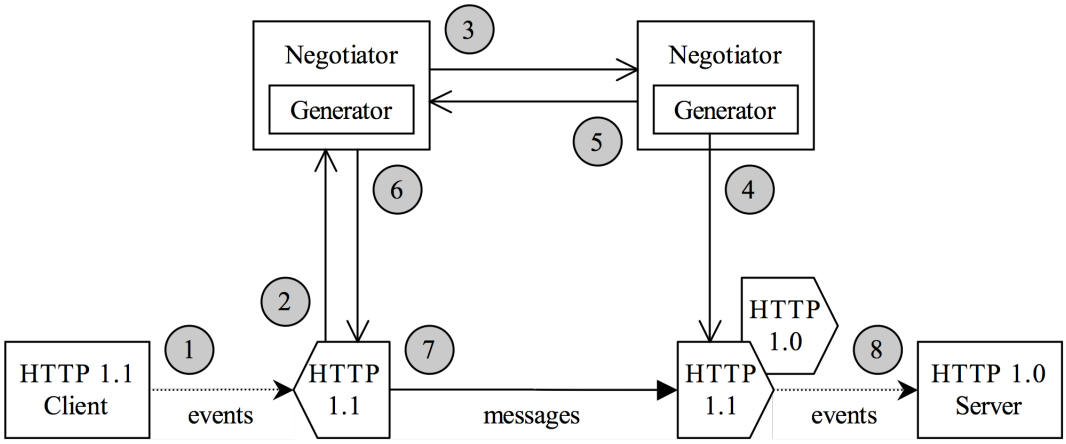

Figure 55: HTTP application scenario using the prototype

The client sends (1) events to its translator, which requests (2) the address of the server-side HTTP 1.1 translator from the client-side negotiator. The client-side negotiator requests (3) this information in turn from the server-side negotiator. The server-side negotiator, finding that no HTTP 1.1 translator exists, decides to generate one, using the specification given by the client-side negotiator. (The specification conceivably can be obtained from elsewhere, such as at a given URI, and while an interesting aspect of the approach, it is not central to the current discussion.) An HTTP 1.1 translator is generated (4) for the server, and its address is returned (5, 6) to the client-side translator via the client-side negotiator. The client-side translator composes an HTTP 1.1 GET request using the information given to it by the client, and the request is sent (7) to the server-side HTTP 1.1 translator for parsing. While the message is parsed, events are forwarded (8) to the server, indicating the concepts encapsulated by the message in the context of the interaction. Although not shown in the figure, an analogous process is used to send an HTTP 1.1 “status 200” response

message to the client; no negotiation is required in this case, since the path is already established.

For this HTTP interaction (and, in fact, most HTTP interactions), only three states are required: one state prior to the request being communicated, one state after the request is communicated and prior to the response being communicated, and one state after the response is communicated. Since we have two roles (client and server), we must have two specifications for the interaction. Figure 56 shows the graphical representation of the interaction specification from a global perspective and from the perspective of both the client role and the server role. Although in this case the specifications are quite similar, this will not be true in general, especially where there are more than two distributed components.

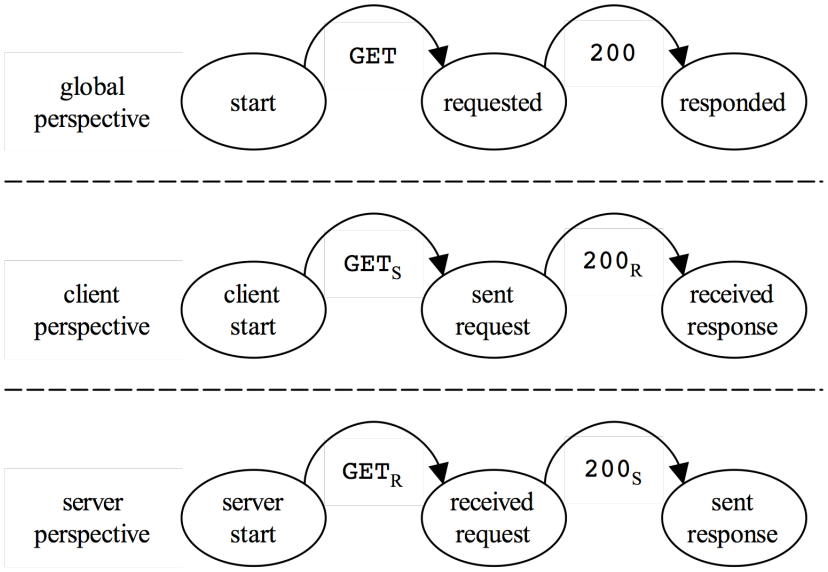

Figure 56: HTTP GET interaction specification

We now provide a bit more detail about the actions taken during the scenario execution. Interpreting the specification of HTTP 1.1, we find that the minimal GET request will require the following events from the client:

```
[01] serl.cs.colorado.edu:80?org.w3c.http/1.1#SERVER
[02] tkn METHOD_GET
[03] tkn URI "/"
[04] end
```

The events are numbered according to the order in which they are given to the translator, although only the position of the last is important. Event 1 gives the address (host and port number) of the server-side negotiator and the role of the server, and is actually provided when the connection to the translator is established. Notice that, since we are using the first of the configurations described in Section 3.3, the well-known port number for HTTP, port 80, is assigned to the server-side negotiator, while the translators are assigned dynamically determined port numbers. This allows the server to service two versions of HTTP at the same well-known port. In contrast,

the other two configurations would require two different port numbers, one for each version of HTTP.

Event 2 indicates to the translator that it will compose a GET request; a parameterized value is not necessary, since it is implicit. Event 3 indicates the target URI of the request (the literal value in quotes being the URI). Event 4 indicates that the client has no more information for the translator, and that the translator should thus attempt to progress through the current interaction.

Having determined that a complete message can be composed from the information provided by the client, the translator requests from the negotiator the address (host and port) for a corresponding HTTP 1.1 translator on the server. Since the negotiator does not have such information, it contacts the server-side negotiator using the given host and port. The server-side negotiator also does not have the information, since no server-side HTTP 1.1 translator exists. The server-side generator is therefore instructed to create one (using JavaCC for the parser specification, *q.v.*, Section 6.2). If the generator does not have the specification for HTTP 1.1, then it can request individual files of the specification from the client-side HTTP 1.1 translator, via the negotiators. Regardless, the specification is used to generate a new translator, and the address at which the translator is listening for incoming messages is communicated back to the client-side HTTP 1.1 translator.

The client-side HTTP 1.1 translator then composes the following HTTP 1.1 GET request, using both information given by the client and defaults determined from the protocol specification:

```
GET / HTTP/1.1
Host: shield.cs.colorado.edu
```

The message is sent to the server-side HTTP 1.1 translator. The client-side translator gives the client the following event:

```
itr { REQUEST_GET start request_sent }
```

This event notes the state change in the data structure used to represent the interaction state from the client-side perspective. As the server-side HTTP 1.1 translator parses the protocol message, the following events are generated and given to the server:

```
[01] tkn METHOD_GET = "GET"
[02] tkn URI = "/"
[03] tkn VER_HTTP = "HTTP"
[04] tkn VER_MAJOR = "1"
[05] tkn VER_MINOR = "1"
[06] tkn VER = "HTTP/1.1"
[07] stc GET_LINE
[08] tkn HEADER_HOST_NAME = "Host"
[09] tkn HEADER_HOST_VALUE = "shield.cs.colorado.edu"
[10] tkn HEADER_HOST = "Host: shield.cs.colorado.edu"
[11] stc GET_HEADER
[12] stc GET_HEADER_LIST
[13] stc GET_REQUEST
[14] msg REQUEST_GET
[15] itr { REQUEST_GET start request_received }
```

The events are numbered according to the order in which they are generated, and labeled according to the part of the protocol from which they result. For example, event 1 is a token event that indicates the recognition of the `GET` method keyword. Note that some token events contain values of nested tokens, which indicates their composite nature. Event 7 is a message structure event that indicates the recognition of a line in a GET request. The six prior events indicate the recognition of tokens that constitute the line. Event 14 is a message event that represents the full message, and event 15 is an interaction event that indicates a change in the interaction state from the server-side perspective.

The server will likely not understand events 8-10, since they are concepts outside the server-side context of HTTP. (Recall that the server is specific to HTTP 1.0 and has not been modified to accommodate HTTP 1.1 concepts.) Regardless, all events will be sent to the server-side event handler, and the server may choose to ignore them or process them as desired; in the case of our simple server, the unrecognized events are ignored, and thus the concepts associated with the "Host" header will be filtered out.

The server-side HTTP 1.1 translator is now finished parsing the GET request, and the last event should have indicated to the server that it has been provided a collection of concepts that comprise a full HTTP message. This completes the process of sending an HTTP request from the client to the server. All construction and parsing was handled by the translators so that the client and server did not need to be concerned with protocol syntax and were only required to handle events that represented concepts of the protocol.

The next step is for an HTTP response to be sent from the server to the client, which would follow an analogous process. No negotiation is required for the return path, since the connection between the translators is still open. Note, too, that the creation of the server-side HTTP 1.1 translator is also a one-time cost; it need not be generated again for any other HTTP 1.1 messages the client, or even a different HTTP 1.1 client, might send.

The configuration of translators and other components of the prototype is just one possible way of employing our approach. For example, one could imagine an implementation that coalesces the functionality of some of these elements, making the

deployment of the implementation perhaps a less daunting prospect. Exploring these options is one thread of our future work.

6.2. Languages and Tools

The development of the prototype went through three iterations, roughly corresponding to the utilities used. As each iteration progressed, certain tools were identified as unsuitable for the development activity for one reason or another, until the current version was constructed. This section will overview the utilities used for prototype implementation, which include a selection of programming language and a compiler compiler (to ease the generation of a translator's parser). Note that an additional tool, namely a finite state machine generator, was considered, though the implementation of such is reasonably trivial and consequently it was determined that attempting to interface with a pre-built tool would be more difficult than implementing one.

The first attempt at a prototype involved the use of C++ [Str97] and the compiler-compiler specification languages of Lex and YACC [LMB92]. This was considered a desirable platform due to the relative efficiency of executable code compiled with the gcc compiler[11]. Automatically generated parsers can already be notoriously slow (top-down parsers, in particular), and it was thought that using C++ over an interpreted language such as Java, in addition to the selection of an LALR parsing utility such as YACC, would provide a reasonably efficient implementation for testing.

---

[11] http://gcc.gnu.org/

However, it was later discovered that a *k*-lookahead parser was necessary (due to the potential for one of several messages to be received, *q.v.* Section 5.2.2), something that is not provided by the standard features of YACC. Additionally, YACC generates LALR(1) parsers, and the checks for conversion and validation of a message specification as restricted by LL(1) needed to be automatically included in the (generated) YACC specification of a message. Although the validations are trivial once the conversions have been performed, the generation of messages at the proper point during a parse is possibly made difficult by such conversion. Further, YACC—as an LR parser generator—generates parsers that perform right-most derivations, which have the possibility to produce semantic concept events in a non-intuitive order. Lastly, multi-threaded translators were designed, and it was later discovered that the Pthreads package as implemented for Mac OSX version 10.2 (the development machine) did not properly implement threads according to the POSIX specification [But97], particularly with respect to locks and condition variables.

Attention then turned toward Java as a development platform, and antlr[12] was briefly selected as the compiler compiler. For our purposes, the primary benefit provided by antlr over YACC was the use of LL(*k*) parsing, something made relatively efficient by antlr's creator, Terry Parr [Par93]. Although this tool is suitable for the prototype, automatically generating an antlr specification (for the translator's parser) was determined to be more difficult than generating a specification particular to other tools. Additionally, we only need one point during a parse with lookahead $k > 1$ (*viz.*, at the very beginning of a message, when it is unknown which message

[12] http://www.antlr.org/doc/index.html

will be parsed). The lookahead depth for an antlr specification applies to the entire grammar, something that was not desired.

JavaCC[13] was then settled on as the compiler-compiler tool. In addition to the benefits of LL(*k*) parsing, the language of a JavaCC specification was determined to be more conducive for automatic generation, and a $k > 1$ lookahead could be specified for only the beginning of a parse.

Thus, the prototype was implemented on a Mac OSX version 10.2 (and subsequently 10.3) machine using Java 1.4.2 as the implementation language, integrating JavaCC version 3.2. As mentioned, no separate utility was used for finite state machine generation.

6.3. Structure

The architecture of the prototype follows from the logical architecture diagrams presented so far (for example, Figure 4 on page 11), the primary difference being that the generator component is integrated as part of the negotiator. The arrangement of components in the prototype architecture is shown in Figure 57.

[13] https://javacc.dev.java.net/

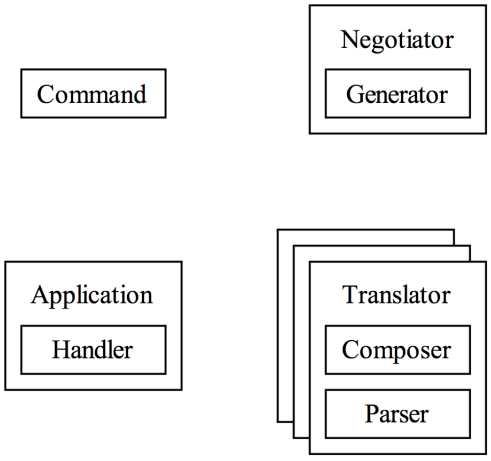

Figure 57: Prototype architecture

Each of the four primary boxes in the diagram represents a separate process. The command process is simply used as a means to start and stop a negotiator and the translators it manages. The application process represents the distributed application component. The negotiator process and the translator process, collectively, comprise the middleware layer between the application process and its corresponding distributed components. (Note that there may be more than one translator process, depending on which translators have been generated and instantiated by the negotiator/generator.) The internal boxes represent the primary subcomponents of the processes. Notice that no interaction state tracker or "interactions" component is shown within the translator; this is due to the fact that interactions are instead maintained as a data structure rather than a logical component (though the distinction is, admittedly, one of convenience).

Of course, these components require some manner of coordination; this is accomplished by a standard socket mechanism. Consequently, the different processes

(except the command process) must be listening for connections. The listener ports and possible connections are enumerated by Figure 58.

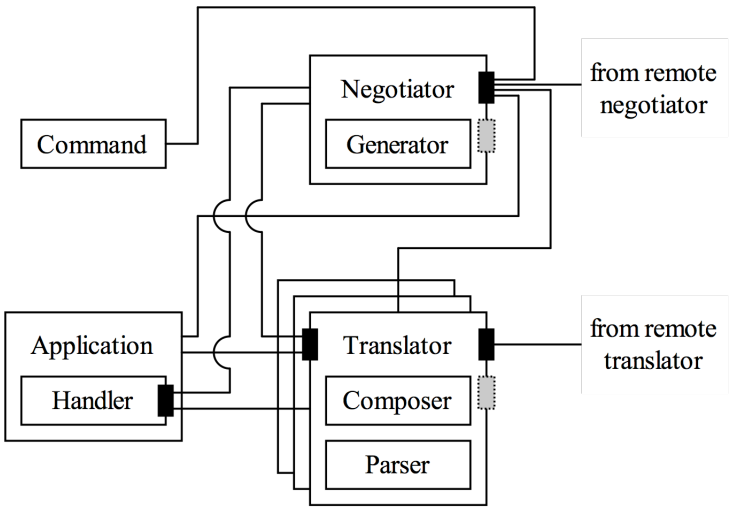

Figure 58: Prototype connections

First, note the "shaded" server sockets of the negotiator and translator. These indicate additional listener ports of the same type as those above them; for the negotiator, this indicates an additional negotiation port; for the translator, this indicates an additional role-specific port.

The connection from the command process to the negotiator is used to initiate commands to the negotiator after it is started. Currently, the only command that can be issued is one to shut down the negotiator and its managed translators, though others (such as shutting down an individual translator) could be easily envisioned and implemented.

The connection from the application to the negotiator allows the application to register its handler as implementing a specific role. The connection opened by the

application to the translator, on the other hand, is used to establish a session in order to initiate (or continue) an interaction.

The negotiator can make two connections to (conceptually) local components: one to the application's handler and one to the translator; both are used during negotiator shutdown. The one to the handler is intended as a means to forcibly unregister the handler from any roles it implements. The one to the translator is intended as a means to shut down the translator. The connection to the translator can also be used to request a role-specific port from the translator. A third connection that a negotiator can open, this one to a (conceptually) remote component, is one with another negotiator; this connection is shown only in terms of the negotiator accepting the connection.

The translator can also make two connections to local components: one to the application's handler and one to the negotiator. The one to the handler is used to establish a session with the handler during an interaction, so that when a translator receives a connection for a particular role the corresponding application handler can be contacted; semantic concept events will then be sent to the handler via this connection. Like the negotiator, the translator can establish a third connection, this one to a remote translator, though again this is shown only in terms of accepting the connection; this connection is used to send the protocol messages.

Finally, we note that the prototype uses a standard architecture for managing connections, in that a separate thread is spawned to manage each accepted connection (though this is not required by the application handler). So, for example, a connection established by a remote translator using one of the role-specific ports will be managed

by a separate connection within the translator, so that other connections will not be refused while a single interaction is being performed.

6.4. Artifacts and Meta-Generation

The prototype is specific to the protocol specification languages developed and defined in Appendix B. Consequently, it must be capable of parsing a protocol specification written using these languages. Rather than constructing a parser by hand for each, a natural choice is to use the same tool used to generate the parser component of a translator; in this case, the tool is JavaCC.

So, we use specifications of the specification languages and a compiler-compiler utility to (partially) generate the prototype. The prototype uses languages written according to the specification languages and the same compiler-compiler utility to generate translators. The translators use messages of a protocol specification written according to the specification languages to generate events and *vice versa*. We have attempted to depict the first part of these verbal acrobatics by Figure 59, which shows the major artifacts in relation to one another during the construction and execution of the prototype.

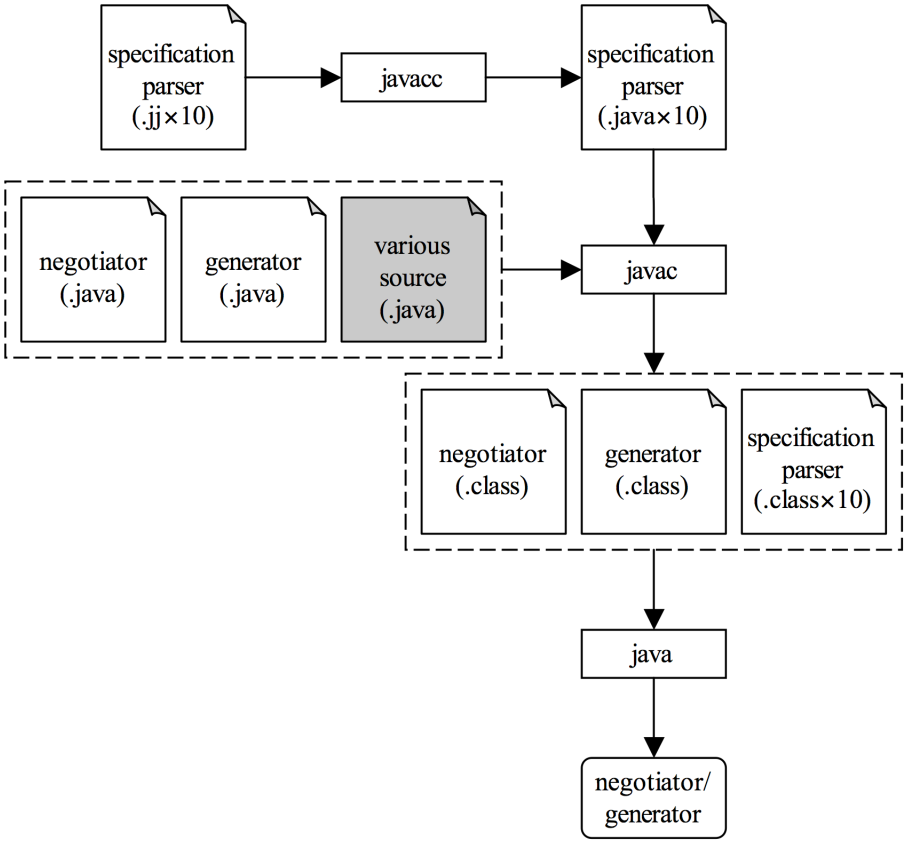

Figure 59: “Development-time” prototype construction

The second part of this process, that of translator generation and operation, is shown by Figure 60. Note that these two figures are connected by the common “negotiator/generator” component (shown in both figures).

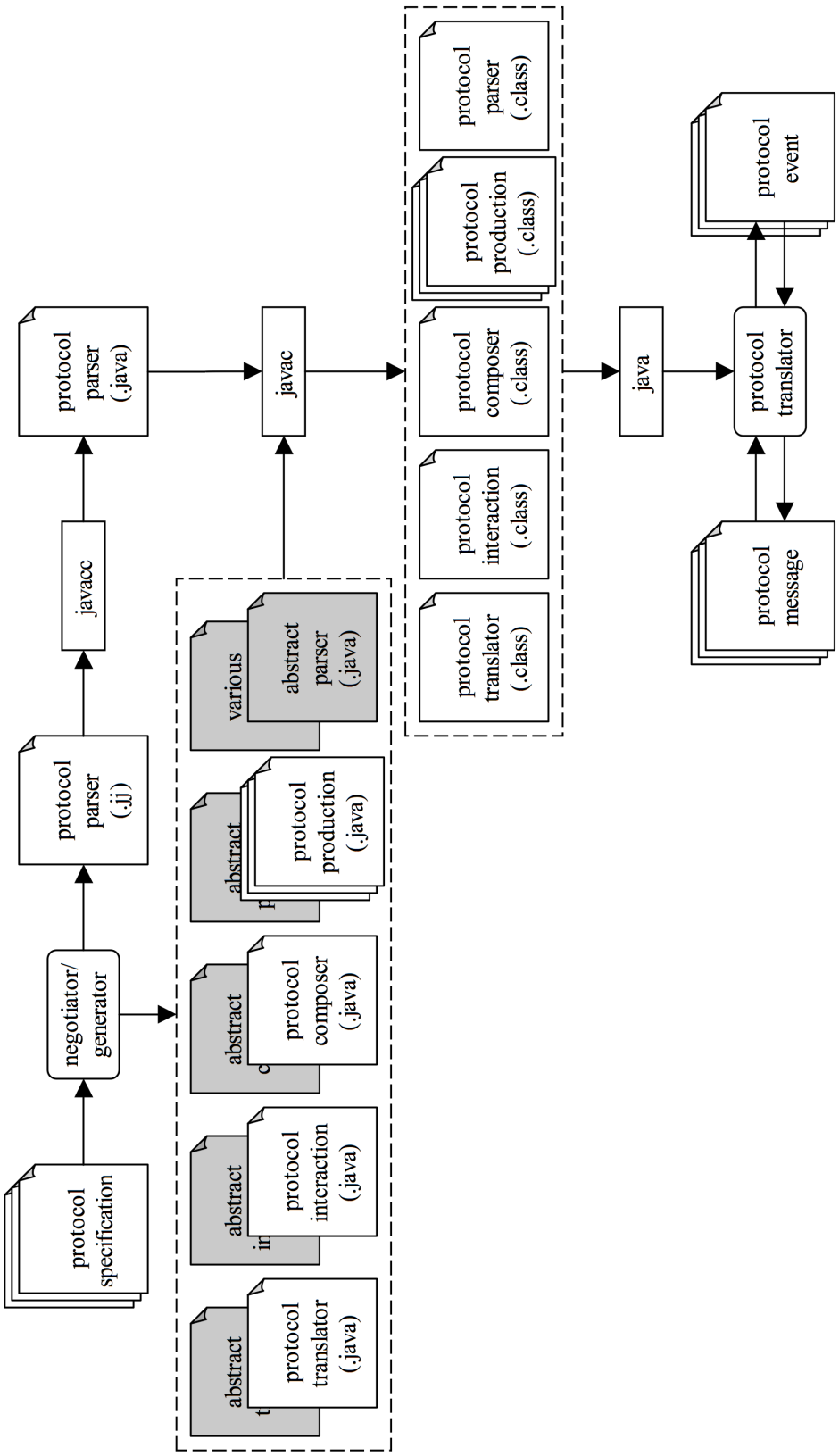

Figure 60: "Run-time" prototype construction

The arrows of the two figures simply indicate data input and output to the processes. Tabbed boxes indicate data; data labeled as ".jj" are JavaCC source files, data labeled as ".java" are Java source files, and data labeled as ".class" are Java bytecode files. (The shaded data boxes represent data artifacts that are secondary to the understanding of the procedure.) Squared boxes indicate generic processes, and rounded boxes indicate processes specific to the event-based translation system. Notice the similarity of operation in the two figures: Each uses a specification to generate the source (partially generate, in the case of the former diagram), which is then used to compile an executable (or in the case of Java, bytecode), which is then executed (or interpreted). This will be important when examining the errors that could be produced by the prototype during operation, outlined in Section 6.6.

6.5. Operation

The operation of the prototype follows the specifications outlined in Chapter 3 and Chapter 5. In this section we will detail the abstract operation of these components in conjunction with one another, whereas Section 6.1 guided the reader through a more concrete example.

Recall that, by our definitions, a "client" is merely a (conceptual) component that initiates a connection with a server, and a "server" is merely a (conceptual) component that waits for a connection to be initiated by a client.

6.5.1. Client-Side Operation

We begin by elaborating on the client-side operation. We assume the following components are available (running): a "client" distributed application component that is aware of the event-based translation system, and a negotiator. The

mechanics are consistent with the abstract client-side coordination, depicted by Figure 15 on page 43.

The application begins by contacting the negotiator at a known port, requesting the application-side port of a translator for a given protocol. The negotiator will search its data directory for the specification of the protocol in question, and, if found, will generate a translator specific to the protocol. (If the protocol specification is not found, an error code is returned to the client.) The translator begins by reporting its application-side port to the negotiator via the standard input/output streams. The negotiator can then return the requested port to the client.

Once the client has obtained a contact port for the translator, it can establish a connection. The client reports its role to the translator, followed by the address (host and port number) of the remote negotiator that acts as the contact point for the server role, followed by the server role. The client now waits for an acknowledgement from the translator.

While the client waits, the translator establishes a connection to its managing negotiator and sends the information about the remote server (host, port number, and protocol-specific role). The client-side negotiator connects to the server-side negotiator, and, by sending the server role, implicitly requests a role-specific port for the corresponding translator. In this way, *the "negotiation" of which protocol and which version to use is always determined by the client*. While the server-side negotiator generates the translator, it will request any missing files of the protocol specification from the client-side negotiator. The server-side negotiator will finish by

sending either an error code or the address of its translator to the client-side negotiator, which can then forward this information back to the translator.

The client-side translator has now been provided with the address of the remote translator to which it will ultimately send protocol messages. It now acknowledges its readiness to the client application, which proceeds by communicating concept events (possibly with associated literal values). These events are added to an abstract syntax tree being maintained by the translator. Once the translator is informed that the client has no more information, it will attempt to compose a message, open a connection to the remote translator, and send the message. This channel remains open until the interaction times out or is determined to have ended.

Note that the prototype performs no optimization. This procedure must be followed each time an interaction is initiated by a client. The only data that is cached are the translator application-side ports, and only by the negotiator managing those translators. Several optimizations (*e.g.*, one translator caching the role-specific address for a remote translator) and parameterizations (the duration a connection stays open before timing out) are possible for future implementations.

6.5.2. Server-Side Operation

For server-side operation, we assume the following components are available (running): a "server" distributed application component that is aware of the event-based translation system, and a negotiator. The mechanics are consistent with the abstract server-side coordination, depicted by Figure 16 on page 47.

The server begins by connecting to the negotiator and registering the address (host and port number) of its handler as implementing a role used by one or more protocols. The server can register as many handlers/roles as desired. Note that the specification for protocols that utilize these roles need not be present in the data directory of the negotiator.

The handler of the server then waits for a connection to be established. The server-side negotiator does likewise. When the negotiator does accept a connection, it first receives a request for a role-specific port of a translator for a particular protocol. Checking its register, the negotiator determines if any server components have listed themselves as implementing the role in question. If not, an error code is returned to the remote, client-side negotiator. If a corresponding server component is instead determined, a translator for the protocol is generated. The translator returns its application-side port to the negotiator via the input/output stream.

Once the translator has been generated and the appropriate port received by the negotiator, the negotiator contacts the new translator at the address and requests that a server socket be created for the role in question. The port of the server's handler is provided as the role's implementer. The translator creates the socket, which is now listening for incoming (protocol) messages. The address of the socket is returned to the negotiator, which forwards it to the client-side negotiator. Now, the handler of the server, the server-side negotiator, and the server-side translator are all waiting for connections. (In the case of the negotiator, it is waiting for another connection, in reference to another interaction).

When the server-side translator accepts a connection at the server socket (specific to the role in question), it connects the input stream of the new connection to a parser. The output of the parser is connected to the handler (listening at the address associated with the server socket, provided by the negotiator), and parsing commences. Generated events are sent to the handler for processing, and a new ACT (Abstract Composition Tree) is instantiated, ready to construct and compose any return message.

Again, the prototype performs no optimization; these actions occur each time an interaction is initiated by a client, with the exception of translator generation, which need be performed only once per version of a protocol. Many optimizations and parameterizations are again possible, though they are left for future implementations.

6.6. Errors

The discussions presented by the last two sections introduce the possibility for several points at which errors can occur. We will categorize the errors that might occur as those that can occur during the generation of a translator and those that can occur during the process of event-based translation.

6.6.1. Errors of Generation

Figure 59 and Figure 60 detail the artifacts used to construct the elements of the event-based translation system; essentially, the former diagram shows the "development-time" half of this construction and the latter diagram shows the "run-time" half of this construction.

For the event-based translation system to function at all, the static process must be correct. Although errors can (and certainly did) occur during development, we will assume a valid negotiator is the output of this half of the construction effort. All of the first-level inputs (i.e., those data elements that are not the output of a process, *viz.*, the JavaCC specification files of the specification parsers, and the negotiator, generator, and various other Java source files) were controlled by the prototype developer and are considered to be correct for the purpose of evaluating the dynamic process.

Prior to the translation of messages and concept events, the latter half of the construction process relies on a single, first-level input (*viz.*, a protocol specification). This specification is not controlled by the prototype developer, but is instead controlled by the protocol/application developer. The prototype developer cannot thus guarantee that any given protocol specification is correct. The processing of such can produce errors at several points along the process of instantiating a translator.

The first errors can occur while the specification parsers of the negotiator are reading the protocol specification. Since a protocol specification is provided piecemeal by small files that reference one another, an entire protocol specification is first read by the negotiator's generator into a single data structure, prior to being operated upon. Syntactic errors within the protocol specification will be detected at this point, and can be easily reported.

After the full specification is read, references within the data structure are resolved. During this phase, some semantic errors of protocol specification are

detected (e.g., a production referencing another, nonexistent production). This is the second level of errors that will be detected.

Once some elementary semantic analysis has been performed, the internal data structure that represents the protocol specification is written out to a JavaCC specification file, which is used as input to the JavaCC utility. Here, the third level of errors will be detected by the compiler compiler, as it performs its analysis of the parser specification file to ensure an LL(1) specification (e.g., unique left prefixes, etc.). Note that the syntax of the parser specification file should be correct, given that it was generated by the event-based translation system.

The remainder of the translator generation should be free of further errors, given that the generated artifacts are correct not only in terms of Java syntax but also with respect to the restrictions imposed by the event-based translation system. Consequently, the execution of the Java compiler (to compile the translator) and the Java Virtual Machine (to execute the translator) should occur normally. An error at this point indicates a problem not with the generated source but with the component and/or process by which that source was generated (i.e., the negotiator's generator).

6.6.2. Errors of Translation

Once a translator has been generated and instantiated, it can receive/send protocol messages and receive/send protocol concept events. Although the outgoing data is controlled, the incoming data is not and is thus another possible source of errors, albeit errors of translation.

Presuming that a corresponding composer of a translator for the appropriate protocol and version is responsible for composing any incoming messages (i.e.,

incoming messages are composed by a remote instance of the event-based translation system), a translator's parser should not encounter an error during the parsing of a message. If, however, the message comes from an independent application without the intermediary of a translator (or was improperly composed, see below), it may not conform to the specification the application purports to implement. In this case (for the prototype), the parse will merely fail and the interaction will terminate abnormally.

Events handed to a translator for message composition can also be erroneous, or at least inconsistent. These inconsistencies can be defined away by the specification of the ACT construction activity, however (*q.v.*, Section 5.3.5). The prototype conforms to the default composition variants, which simply ignore the possibility of inconsistency and are thus capable of producing erroneous messages under unusual circumstance of conflicting input.

7. Case Studies

This chapter will discuss the experience that the prototype has given us with respect to protocol specification and translation. Through the description of several experiments and their results, we will argue that the feasibility of the approach has been demonstrated. We will also present several issues that were raised by the experiments, issues that highlight limitations of the approach and suggest restrictions for protocol specification.

Three protocols were initially targeted: HTTP (version 1.0 [BFF96] and version 1.1 [FGM+99]), SIENA (version 1.4 and version 1.5) [Car98], and Weblogs (the Blogger API[14] and the MetaWeblog API[15]). HTTP and SIENA turned out to be good targets for experimentation. The Weblogs interfaces, on the other hand, do not define protocols of their own, and instead use XML-RPC[16] as a carrier.

The sections on HTTP and SIENA will first present the protocol specifications, followed by descriptions of the experiments performed and the lessons learned from these experiments. The section on Weblogs will instead discuss why Weblogs, not

---

[14] http://www.blogger.com/developers/api/documentation20.html
[15] http://www.xmlrpc.com/metaWeblogApi
[16] http://www.xmlrpc.com/spec

being a protocol, was an inappropriate target from the outset, though nevertheless useful in describing some limitations of the system.

7.1. HTTP

The first protocol to be studied was HTTP. Much of the dissertation proposal was focused on using HTTP as an example, and consequently the protocol itself was already well studied. The goals for the HTTP 1.0 and HTTP 1.1 specifications used by the prototype were not to evaluate HTTP as a protocol, but rather to begin to define the limits of what aspects of a protocol could be reasonably handled by the prototype.

The following subsections will explore the specifications of HTTP used for the initial prototype evaluation, the arrangement of the particular experiments performed, and the lessons learned from those experiments.

7.1.1. Specification

Because HTTP as a protocol was not being evaluated, only some elements of the protocol were considered. Specifically, we classified elements of the protocol such that representative elements could be selected and specified in the languages developed for this work. For HTTP 1.0, these elements included: two complete interactions, two request messages (GET and POST), two response messages (status 200 and status 404), the elements of the request line, the elements of the status line, a header that is not relevant to context-sensitive termination, a header that is relevant to context-sensitive termination, and a message body. This leaves several interactions and several messages unspecified, but since other interactions are identical in the sense that they are defined by a single request-response, and since

each message has a unique, required token (i.e., for request messages the method name, and for response messages the status code), no ambiguity can possibly result during composition, parsing, or interaction state tracking. This also leaves several headers unspecified, but these are analogous to the representative (non-termination related) header, all of which are technically optional; the specification of additional headers would thus add nothing to our functional evaluation.

The HTTP 1.0 specification is spread over several files according to the definitions of the specification languages and according to logical groupings. However, for convenience they have been replicated below as Figure 61, Figure 62, and Figure 63, for the interaction, the GET request message (the POST message is not shown), and the status 200 response message (the status 404 response message is not shown), respectively. Note that concept specifications are also not shown, as these should be obvious from the specification (i.e., any name to the right of an equals sign represents an association with a concept that must be specified).

```
[initializations]
  CLIENT START
  SERVER START
[communications]
  REQUEST_GET
     : {START} CLIENT REQUESTED
    -> {START} SERVER REQUESTED
  RESPONSE_200
     : {REQUESTED} SERVER RESPONDED
    -> {REQUESTED} CLIENT RESPONDED
```

Figure 61: HTTP 1.0 GET interaction

```
[tokens]
  URI ("/") = URI
  SPACE (" ")
  CRLF ("\r\n")
  METHOD_GET_ID ("GET") = METHOD_GET_NAME
  PROTOCOL_ID ("HTTP")
  PROTOCOL_VERSION_MAJOR_ID ("1") = VERSION_MAJOR
  PROTOCOL_VERSION_MINOR_ID ("0") = VERSION_MINOR
  PROTOCOL_VERSION (
    <PROTOCOL_VERSION_MAJOR_ID> "."
    <PROTOCOL_VERSION_MINOR_ID>
  )
  PROTOCOL (<PROTOCOL_ID> "/" <PROTOCOL_VERSION>)
    = PROTOCOL
  SEP (": ")
  CONTENT_TYPE_ID ("Content-Type")
  CONTENT_TYPE_VALUE ("text/plain") = CONTENT_TYPE
  CONTENT_LENGTH_ID ("Content-Length")
  CONTENT_LENGTH_VALUE ("0" | ["1"-"9"](["0"-"9"])*)
    = CONTENT_LENGTH
[structures]
  CONTENT_TYPE (
    <CONTENT_TYPE_ID> <SEP> <CONTENT_TYPE_VALUE
  )
  CONTENT_LENGTH (
    <CONTENT_LENGTH_ID> <SEP> <CONTENT_LENGTH_VALUE>
  )
  HEADER (CONTENT_TYPE | CONTENT_LENGTH)
  HEADER_LIST (HEADER <CRLF>)*
[message]
  LINE (
    <METHOD_GET_ID> <SPACE> <URI> <SPACE> <PROTOCOL>
  )
  REQUEST_GET (LINE HEADER_LIST <CRLF>)
    = REQUEST DOWNLOAD
```

Figure 62: HTTP 1.0 GET message

```
[tokens]
  SPACE (" ")
  CRLF ("\r\n")
  STATUS_200_ID ("200") = STATUS_200_NAME
  PROTOCOL_ID ("HTTP")
  PROTOCOL_VERSION_MAJOR_ID ("1") = VERSION_MAJOR
  PROTOCOL_VERSION_MINOR_ID ("0") = VERSION_MINOR
  PROTOCOL_VERSION (
    <PROTOCOL_VERSION_MAJOR_ID> "."
    <PROTOCOL_VERSION_MINOR_ID>
  )
  PROTOCOL (<PROTOCOL_ID> "/" <PROTOCOL_VERSION>)
    = PROTOCOL
  REASON_PHRASE ("OK")
  SEP (": ")
  CONTENT_TYPE_ID ("Content-Type")
  CONTENT_TYPE_VALUE (“text/plain”) = CONTENT_TYPE
  CONTENT_LENGTH_ID ("Content-Length")
  CONTENT_LENGTH_VALUE ("0" | ["1"-"9"](["0"-"9"])*)
    = CONTENT_LENGTH
[structures]
  CONTENT_TYPE (
    <CONTENT_TYPE_ID> <SEP> <CONTENT_TYPE_VALUE
  )
  CONTENT_LENGTH (
    <CONTENT_LENGTH_ID> <SEP> <CONTENT_LENGTH_VALUE>
  )
  HEADER (CONTENT_TYPE | CONTENT_LENGTH)
  HEADER_LIST (HEADER <CRLF>)*
[message]
  LINE (
    <PROTOCOL> <SPACE> <STATUS_200_ID> <SPACE>
    <REASON_PHRASE>
  )
  RESPONSE_200 (LINE <CRLF> HEADER_LIST <CRLF> <BODY>)
    = RESPONSE SUCCESS
```

Figure 63: HTTP 1.0 status 200 message

Note that the specification for the request and the specification for the response share many elements; in the actual specification, these are abstracted such that they are specified only once.

For HTTP 1.1, the representative elements include all those of HTTP 1.0, in addition to several elements that highlight the major differences between the two protocols, which include modifications to the GET interaction and the POST

interaction, the specification of an additional message, and the specification of an additional (required) header. Again, this leaves several interactions, several messages, and several headers undefined, though for similar reasons these fall into the same equivalence classes as the elements selected for specification.

The HTTP 1.1 specification is also spread over several files. However, the specification is reproduced below (less the concept specifications, which again should be evident) as Figure 64, Figure 65, and Figure 66. As mentioned, our specification of HTTP 1.1 is a superset of our specification for HTTP 1.0, so those elements common to both versions will not be replicated.

```
[initializations]
  CLIENT START
  SERVER START
[communications]
  REQUEST_GET
     : {START} CLIENT REQUESTED
    -> {START} SERVER REQUESTED
  RESPONSE_100
     : {REQUESTED} SERVER REQUESTED
    -> {REQUESTED} CLIENT REQUESTED
  RESPONSE_200
     : {REQUESTED} SERVER RESPONDED
    -> {REQUESTED} CLIENT RESPONDED
```

Figure 64: HTTP 1.1 GET interaction

```
[tokens]
  SPACE (" ")
  CRLF ("\r\n")
  STATUS_100_ID ("100") = STATUS_100_NAME
  PROTOCOL_ID ("HTTP")
  PROTOCOL_VERSION_MAJOR_ID ("1") = VERSION_MAJOR
  PROTOCOL_VERSION_MINOR_ID ("0") = VERSION_MINOR
  PROTOCOL_VERSION $constant (
    <PROTOCOL_VERSION_MAJOR_ID> "."
    <PROTOCOL_VERSION_MINOR_ID>
  )
  PROTOCOL (<PROTOCOL_ID> "/" <PROTOCOL_VERSION>)
    = PROTOCOL
  CONTINUE_PHRASE ("CONTINUE")
[message]
  LINE (
    <PROTOCOL> <SPACE> <STATUS_100_ID> <SPACE>
    <CONTINUE_PHRASE>
  )
  RESPONSE_100 (LINE <CRLF> <CRLF>) = RESPONSE_CONTINUE
```

Figure 66: HTTP 1.1 status 200 message (differences from HTTP 1.0)

Notice the change to the GET interaction, which allows the server to send an arbitrary number of status 100 responses to the client. This is an important difference between the two protocols that we must resolve automatically. Additionally, the "Host" header is not only a new header, but also mandatory in the request. An application that does not understand this header must still be capable of composing, via the prototype, an HTTP 1.1 GET request. (However, observe that the value of the `HOST_VALUE` token is the empty string; the reason for this blank value is explained in the next subsection.)

Of course, there are many other differences between HTTP 1.0 and HTTP 1.1, some of which are truly semantic ("concept") changes requiring modifications to HTTP 1.0 servers (e.g., the "Connection" header describes when a connection should now be closed, since multiple interactions are permitted over the same connection).

Our aim is not to address HTTP evolution specifically, but rather to illustrate the feasibility of dynamic protocol evolution with a well-known protocol.

7.1.2. Setup

Each experiment performed using the HTTP specifications used an identical setup, according to an arrangement similar to that shown in Figure 55 on page 175. Below is a table summary of the experiments performed using the HTTP specifications. The table shows the version of HTTP understood by both the client and the server, which messages were sent, and the primary goal(s) of the experiment. Unshaded rows indicate experiments that used the HTTP 1.0 specification, whereas shaded rows indicate experiments that used the HTTP 1.1 specification:

Client Version	Server Version	Messages Set	Goal
HTTP 1.0	HTTP 1.0	GET request	Evaluate prototype framework; test message composition technique
HTTP 1.0	HTTP 1.0	GET request 200 response	Test interaction-state tracking
HTTP 1.0	HTTP 1.0	POST request 200 response	Test interaction-state tracking; test message choice for parsing/composition
HTTP 1.0	HTTP 1.0	GET request 404 response	Test interaction-state tracking; test message choice for parsing/composition
HTTP 1.0	HTTP 1.0	POST request 404 response	Test interaction-state tracking; test message choice for parsing/composition
HTTP 1.1	HTTP 1.0	GET request 200 response	Test extraneous information in message composition
HTTP 1.0	HTTP 1.1	GET request 200 response	Test missing information in message composition
HTTP 1.1	HTTP 1.0	GET request 200 response	Test context-sensitive termination
HTTP 1.0	HTTP 1.1	GET request 100 response 200 response	Test spurious interaction-state tracking with messages unknown to the application
HTTP 1.0	HTTP 1.1	GET request 100 response 200 response	Test spurious interaction-state tracking with messages unknown to the application

The first experiment performed with the HTTP specifications involved the sending of a simple HTTP 1.0 GET request from an HTTP 1.0 client to an HTTP 1.0 server:

```
GET / HTTP/1.0
```

This is the simplest possible HTTP message. (Note that a second carriage-return/linefeed, not shown, follows the line.) While seemingly trivial, this test aided us not only in evaluating the stability of the framework but also in validating the feasibility of our message composition technique. HTTP turned out to be a prime candidate for this simple test, since the protocol is well established, the message

format is simple, text-based, and fixed (versus freeform, requiring whitespace specifications), and the specification could easily be incrementally expanded as further testing was desired.

The next step was to add a status 200 (OK) response message after the GET message was received and processed:

```
HTTP/1.0 200 OK
Content-Length: 0
```

This enabled us to test a translator's ability to manage the coordination among the interaction-state tracking, parsing, and composing activities, according to an interaction specification.

To this point the case study of HTTP was merely an aid to evaluate the correct functioning of the prototype. However, the addition of the return message foreshadowed a $k > 1$ lookahead problem.

The next step was to add the specifications of the POST message, and finally the status 404 message, to the HTTP 1.0 specification. The POST request did not explicitly reveal the lookahead problem, though it was painfully obvious with the addition of the status 404 response. The way in which this was overcome will be discussed in the next subsection.

With four successful exchanges (GET-200, GET-404, POST-200, and POST-404), we continued with the next stage of experimentation, which was to formulate an HTTP 1.0 message from HTTP 1.1 concepts. The faux HTTP 1.0 client was upgraded to HTTP 1.1, so that it would provide HTTP 1.1 concept events to the translator for composition. (Note that, in the current version of the prototype, the client determines which version of which protocol will be used for communication. In this case,

HTTP 1.1 would normally be used, given the preference of the client. However, for the purpose of this experiment, the next experiment, and the last experiment, the version of the protocol in use was manually fixed in contrast to that specified by the client.) Since the translator is generated to ignore extraneous information provided by a connected application, the concept corresponding to the value of the "Host" header was disregarded during composition and a (simple) valid HTTP 1.0 GET or POST message could be composed without difficulty.

Once this simple case of extraneous information was tested, it was time to test the converse: the simple case of missing information. While the minimum HTTP 1.0 GET message requires only the request line with no compulsory headers, the minimum HTTP 1.1 GET message requires at least the "Host" header:

```
GET / HTTP/1.1
Host: serl.cs.colorado.edu
```

The client was downgraded to HTTP 1.0 and the server upgraded to HTTP 1.1. The client, now unaware of the requirement of the "Host" header, failed to provide such information, leaving incomplete the ACT being constructed by the translator's composer. No message could be composed, and the interaction failed. To rectify this situation, we were left with two possibilities: specify the "Host" header as having a default literal value, or offer a mechanism for parameterizing certain tokens of a protocol specification. The former seemed unreasonable, and the latter required modification to the specification languages. In the end, the prototype takes neither of these approaches, instead preferring a hard-coded, inelegant solution, which is to use a value from the properties of the Java runtime environment; this must be fixed for future versions of the prototype.

Ignoring various minor syntax manipulations (such as changing the separator character between a header name and value, adding additional meaningless headers, etc.), the next experiment performed with the HTTP message specifications involved the use of the "Content-Length" field in a status 200 response and a nonempty message body, in order to evaluate a generated parser's ability to detect such context-sensitive termination, in conjunction with the scenario of an HTTP 1.1 client attempting to communicate with an HTTP 1.0 server (described in Section 6.1). This experiment performed successfully, using HTTP 1.1 as the protocol (despite the lesser understanding of the server), though it highlighted the difficulties with the composition of such messages as discussed in Section 5.3.7.

Lastly, a difference in HTTP interactions was exploited to evaluate the translator's ability to deal with syntactic differences in the interaction specifications between two versions of a protocol. As already mentioned, an HTTP 1.1 server can respond to a client with any number of status 100 (continue) messages before the actual response is sent. This is a feature included in HTTP 1.1 for content that may take some time to procure, such as HTML/XML/etc. data formed by various database queries, in order to prevent the connection from timing out. The addition of these messages breaks the traditional request/response paradigm of the protocol, however, and we must now deal with interactions that are more complicated.

This experiment was actually two experiments, as we required the unexpected status 100 messages to be dealt with from either side of the distributed application, depending on the version of the protocol in use. The experiments involved reverting to the HTTP 1.0 version of the faux client and the HTTP 1.1 version of the faux

server; the first experiment used HTTP 1.0 for communication, and the second used HTTP 1.1 for communication. Both experiments performed successfully, as the "100" message data provided by the server was considered extraneous by the composer of the server's HTTP 1.0 translator in the first experiment and by the interaction-state tracker of the client's HTTP 1.1 translator in the second experiment (though the client still received events from the parser regarding the messages, which were ignored).

7.1.3. Results

The results of the HTTP experiments were largely successful, demonstrating the basic functionality of the prototype, including parsing, but more importantly correct message composition and interaction-state tracking.

However, the question of sufficiency of single-symbol lookahead was raised by the addition of the status 200 response message: The parser, previously generated to handle only our simple GET message, now had a choice with respect to which message needed to be parsed, presuming that any message could be received at any point during the interaction. For the experiment in question, this was not actually an ambiguity, since the initial token of each message is unique (*viz.*, "`GET`" for the GET message and "`HTTP`" for the status 200 message). However, early analysis of message choice forced a redesign of the interaction-state tracker, so that the parser would recognize only those message expected at each point during the interaction. For example, since an HTTP server would not expect to receive a status 200 message (or any response message, for that matter), the translator, when accepting a connection at the port associated with the server role, should initialize the parser to a state in which only requests will be recognized as valid messages. Then, even if a

properly formatted status 200 message were sent to the server in place of a request, it would cause a parsing error (rather than being identified as an inconsistency in the interaction state).

The addition of a POST interaction to the HTTP 1.0 specification also did not reveal the lookahead problem, though a simplistic POST message could be both composed and parsed without difficulty, just as with the GET message. It was not until the interactions of the specification were expanded to include a status 404 (not found) response message that the problem with multiple symbol lookahead was identified. Now, when the client was receiving a message in response to either a GET or a POST request, it could be either a status 200 or a status 404 response, both of which begin with the token "HTTP" according to our specification. Not even the refinement of interaction-state tracking that directs a parser to recognize only expected messages was able to solve this ambiguity.

Our experience had already indicated a potentially disastrous problem, one that was overcome by the lookahead specification mechanism of JavaCC. The "start" token of the given state could be assigned a lookahead value greater than 1, in order to identify which message was being parsed, and the remainder of the message could be parsed using a single-token lookahead. However, the question of "what value *k*?" arose.

The lookahead value that allows a parser to determine which message is being parsed is dependent on the number and specification of the potential messages that can be received during any given interaction state. When no message can be received (meaning messages can only be composed and sent), the lookahead value is, of

course, irrelevant. When only a single message can be received, the lookahead value is 1. When more than one message can be received, the lookahead value is determined according to the collective specifications of the messages.

According to our HTTP 1.0 specification, the distinction is identified only after seven tokens are identified. The tokens of the status line for both the status 200 message and the status 404 message are shown below:

1)	`HTTP`	`HTTP`	
2)	`/`	`/`	
3)	`1`	`1`	
4)	`.`	`.`	
5)	`0`	`0`	
6)	(space)	(space)	
7)	`200`	`404`	← **differentiating token**
8)	(space)	(space)	
9)	`OK`	`Not found`	
10)	(CRLF)	(CRLF)	

Of course, another specification might define the entire status line to be a single token, thus removing the ambiguity. However, this would preclude the parser from generating finer-grained concept events. So, we are left with $k \geq 7$ for the HTTP specification. Although this value can be calculated directly from the specification, certain aspects of the protocol specification methodology proposed by this work, particularly with respect to context-sensitive termination and subtoken concept association, can make such a determination difficult. Instead, we are left with the options of (1) allowing the protocol developer to specify an appropriate value or (2) predetermining a fixed value. Not wishing to further alter the specification languages, we opted for the latter possibility, fixing $k = 8$.

The benefit of fixing the value for the HTTP scenarios was that the value is relatively low and did not visibly impact the performance of the parser. The primary

drawback is that all HTTP messages that can be received in the same context must then be differentiable within eight tokens. In fact, all within the HTTP 1.0 specification are, and all but two within the HTTP 1.1 specification are. Those two within HTTP 1.1 that are not differentiable within eight tokens are a GET message and a "conditional" GET message; what differentiates a GET message from a conditional GET message is the absence/presence (respectively) of any of the following headers: "If-Modified-Since", "If-Unmodified-Since", "If-Match", "If-None-Match", or "If-Range". Since these headers can occur essentially arbitrarily deep within the message, an upper bound on the token lookahead cannot be determined. As we discuss in the next section, this was an even larger issue for the SIENA protocol.

Two important (untested) issues were raised by the analysis of HTTP within this specification methodology: First, the timeout problem is not addressed (and, in fact, is exacerbated) by the elimination of these status 100 messages; second, the lack of application coordination with the translator can potentially force an unexpected (and unnecessary) timeout. The former issue can be solved by the addition of timeout specifications to the interaction specification grammar; the second issue is more complex, and requires an application to respond to certain events produced by the translator. If the translator generates an interaction event, recording a change from one state to another, the application must provide a Boolean response, indicating whether it understands the new interaction state. If the application understands, the translator can expect events (possible only an "`end`" event) to be generated by the application; if the application does not understand, the translator can attempt to

progress through the interaction by composing and sending messages automatically, without the intervention of the application, until a known state is reached.

7.2. SIENA

The second protocol to be studied was SIENA. SIENA is a publish-subscribe, content-based routing system composed of a network of servers, subscribers, and publishers. The network of servers is first constructed. Subscribers "subscribe" to the network, requesting to receive notifications that conform to certain constraints (e.g., "price < 100"); these constraints are propagated through the network as necessary. Publishers then "publish" notifications (e.g., "price = 50") to the network; these notifications are routed through the network to the subscribers based on the content of the notifications and the constraint information maintained by the servers.

We are not concerned with the behavior of the network as a whole, but rather the behavior of the individual servers. While one can consider the SIENA protocol to be defined by the performance of the entire network during any given subscription or publication, we need not view things so globally due to the fact that, once a subscription (or a publication) is handed to a server by a component, what that server does with the message is no longer the concern of the component. Instead, we view each exchange between two components as one interaction of the protocol. (In other words, we are concerned only with the protocol at the lowest scope of communication between some set of components of a distributed application.) Thus, during the propagation of a subscription (or a publication) throughout the network, each affected server will act as a server (the component with which an interaction is initiated) and zero or more clients (the component that initiates an interaction). During the

propagation of a subscription, the subscriber will act as a client. During the propagation of a publication, the publisher will act as a client and any subscribers receiving the notification will act as a server. Consequently, the propagation of a message throughout the network (an "interaction" in the larger sense) can be perceived as a chain of interactions (in the smaller sense) between two components. For our experiments, we will construct a simple network of two connected servers, with a subscriber on one "end" and a publisher on the other; this defines a chain of three interactions, though we will concern ourselves only with the interactions that occur between the two servers.

Whereas only representative aspects of the HTTP protocol were specified, the entire protocol of SIENA was specified for both versions. The goals for the specifications of SIENA 1.4 (SIENA 1.4.3, specifically) and SIENA 1.5 (SIENA 1.5.0, specifically) were primarily to experiment with another protocol that might introduce some issues absent in the representative specifications of the versions of HTTP. Additionally, we felt it important to experiment with a fully specified protocol, in order to reveal potential issues with message composition and obtain some reasonable timing metrics.

The following subsections will explore the specifications of SIENA used for further prototype evaluation, the arrangement of the particular experiments performed, and the lessons learned from those experiments.

7.2.1. Specification

Although, like HTTP, SIENA as a protocol was not being evaluated, the entirety of both protocol versions were specified. Specifically, we classified elements

of the protocol such that elements could be specified in the languages developed for this work. For SIENA 1.4, these elements include eleven complete interactions and eleven messages.

The SIENA 1.4 specification is spread over several files according to the definitions of the specification languages and according to logical groupings. For convenience, some of them have been replicated below as Figure 67 and Figure 68, for a representative interaction (a "publication") and for the relevant message (PUB), respectively. Note that concept specifications are not shown, as these again should be obvious.

```
[initializations]
  CLIENT START
  SERVER START
[communications]
  PUBLICATION
     : {START} CLIENT END
    -> {START} SERVER END
```

Figure 67: SIENA 1.4 PUBLICATION interaction

```
[tokens]
  OPEN ("{")
  CLOSE ("}")
  SEP ("=")
  NAME_START (["A"-"Z", "a"-"z", "_"])
  NAME_PART (NAME_START | [".", "/", "$"])+
  NAME (<NAME_START> (<NAME_PART>)?)
  NULL ("null") = NULL_VALUE
  TRUE ("true")
  FALSE ("false")
  BOOLEAN (<TRUE> | <FALSE>) = BOOLEAN_VALUE
  INTEGER ("0" | ["1"-"9"](["0"-"9"])*) = INTEGER_VALUE
  DECIMAL (
    ("0" | ["1"-"9"](["0"-"9"])*) "." (["0"-"9"])+
  ) = DECIMAL_VALUE
  STRING (
    "\""
    ([" "-"!", "#"-"[", "]"-"~"] | "\\\"" | "\\\\")*
    "\""
  ) = STRING_VALUE
  SENP ("senp")
  METHOD_NAME ("method")
  METHOD_VALUE_PUB optional constant ("PUB")
    = METHOD_PUB
  TO_NAME ("to")
  TO_VALUE (<ID>) = TO
  ID_NAME ("id")
  ID_VALUE (<ID>) = ID
  HANDLER_NAME ("handler")
  HANDLER_VALUE (<URL>) = HANDLER
  VERSION_NAME ("version")
  VERSION_VALUE constant ("1.4.3") = VERSION
  EVENT_NAME ("event")
  EVENTS_NAME ("events")
  ATTRIBUTE (<NAME> <SEP> <VALUE>) = ATTRIBUTE
  VALUE (
    <NULL>
  | <BOOLEAN> | <INTEGER> | <DECIMAL> | <STRING>
  )
[structures]
  METHOD_PUB (<METHOD_NAME> <SEP> <METHOD_VALUE_PUB>)
  TO (<TO_NAME> <SEP> <TO_VALUE>)
  ID (<ID_NAME> <SEP> <ID_VALUE>)
  HANDLER (<HANDLER_NAME> <SEP> <HANDLER_VALUE>)
  VERSION (<VERSION_NAME> <SEP> <VERSION_VALUE>)
  EVENT (<EVENT_NAME> <OPEN> (<ATTRIBUTE>)+ <CLOSE>)
  EVENTS (<EVENTS_NAME> <OPEN> (EVENT)+ <CLOSE>)
[message]
  HEADER (
    <SENP> <OPEN> METHOD_PUB TO (VERSION)? <CLOSE>
  )
  PUBLICATION (HEADER (EVENT | EVENTS)) = PUBLICATION
```

Figure 68: SIENA 1.4 PUBLICATION message

For SIENA 1.5, the specified elements include all those of SIENA 1.4, with one important restriction: the deprecation of an initial interaction that exchanged a WHO and an INF message. The SIENA 1.5 specification is also spread over several files in a nearly identical matter to that of the specification of SIENA 1.4.

7.2.2. Setup

Each experiment performed using the SIENA specifications used an identical setup, according to the "double proxy" approach as shown in part (c) of Figure 18 on page 50. Below is a table summary of the experiments performed using the SIENA specifications. The table shows the version of SIENA understood by both the publisher-side server and the subscriber-side server, which messages were sent, and the primary goal(s) of the experiment. Unshaded rows indicate experiments that used the SIENA 1.4 specification, whereas the shaded row indicate experiments that used the SIENA 1.5 specification:

PUB-side Server Version	SUB-side Server Version	Messages Set	Goal
SIENA 1.4	SIENA 1.4	WHO request INF response SUB PUB	Evaluate the SIENA specification and new features of the specification languages; gather timing data
SIENA 1.4	SIENA 1.5	WHO request INF response SUB PUB	Test deprecation of WHO/INF interaction; gather additional timing data
SIENA 1.5	SIENA 1.4	SUB PUB	Test absence of WHO/INF interaction; gather additional timing data

The SIENA experiments differed from the HTTP experiments in that the relevant application components that use the respective protocol are not "aware" components. They are, instead, out-of-the-box SIENA hierarchical dispatchers (i.e.,

SIENA servers), a simple tailored SIENA subscriber, and a simple tailored SIENA publisher. Specifically, the subscriber sends a subscription message to the subscriber-side server, which is propagated to the publisher-side server through two SIENA translators. The publisher then sends a publication (notification) to the publisher-side server, which is propagated to the subscriber-side server through the same two SIENA translators, which is propagated to the subscriber. The route of publications is shown in Figure 69; for simplicity (since we are concerned only with the interactions between the servers), the publisher and the subscriber are not shown.

Figure 69: SIENA 1.4 configuration

Note that the use of negotiators was not necessary, since both translators were considered to be on one "side" (i.e., virtual machine) of a distributed application. In this instance, the translators would be pre-generated and pre-configured, explicitly for the purpose of processing interactions initiated by another SIENA 1.4 server. While this may seem unnecessary, the configuration allows the addition of other "receiver" translators for different versions of the protocol to be added without interruption.

The first experiment performed with the SIENA specifications involved the sending of a publication from one SIENA 1.4 server to another. In order for this to succeed, the servers first needed to "know" about each other via an exchange of WHO and INF messages. This interaction was performed first, the servers communicating these two messages simply. A simple SIENA client then had to

subscribe for the coming publication. A simple publication message followed as a separate, consecutive interaction over a new connection:

```
event {
  name="Nathan D. Ryan"
  age=32
  nationality="US"
}
```

(Note that SIENA messages also include a header that is unique to each type of message; the contents of this header are determined entirely by the SIENA server.)

The second experiment involved an identical setup, except ten thousand publications were sent, instead of only one. This was performed for the purpose of obtaining some relative timing data, in order to determine the drop in efficiency due to the intervening translators. (Consequently, the same ten thousand publications were also sent without the intervening translators, as a point of comparison.)

The next pair of experiments mirrored the first two, though the receiving server was exchanged for a SIENA 1.5 server, and the second (right-hand-side) translator was consequently exchanged for a SIENA 1.5 translator, as shown below in Figure 70.

Figure 70: SIENA 1.4/1.5 configuration

The primary difficulty with these experiments was that SIENA 1.5 is designed as an entirely connectionless protocol, so the WHO/INF exchange of SIENA 1.4 is deprecated. While the second translator could simply dispose of the events so that the WHO message was invisible to the receiving server, the first translator cannot

automatically compose and return the INF message, because the message must reflect information provided in the corresponding WHO message. This stateful aspect cannot be captured by default values, and the second translator, having no knowledge of an INF message, cannot provide the appropriate concepts.

The solution was to insert a "filter" component capable of both recognizing the relevant concepts (in fact, only one concept, *viz.*, a server ID) of a WHO message provided by some translator and passing the appropriate concepts back to that translator for the construction of the INF message. This component allows all concepts to pass through unaltered. The new scenario is shown in Figure 71.

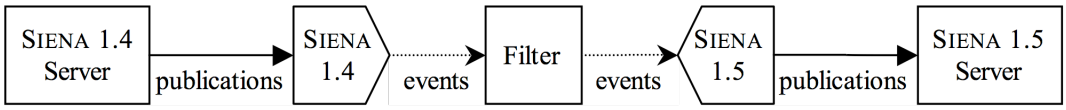

Figure 71: SIENA 1.4/1.5 configuration with filter

This requires that the filter be designed specifically as an intermediary between these two versions of the protocol. It may be that future work will identify a means of generating these intermediaries automatically under certain circumstances, though even this is undesirable because it means a potential filter for every pair of protocols, which is a second order-of-magnitude problem.

The third and final pair of experiments involved a SIENA 1.5 server sending publications to a SIENA 1.4 server, as shown in Figure 72.

Figure 72: SIENA 1.5/1.4 configuration

Note that the filter component was not required for this experiment, because the WHO/INF exchange does not take place.

The final combination of components possible with these versions of the SIENA protocol would link a SIENA 1.5 server to another SIENA 1.5 server. This combination was skipped because not only is it relatively trivial, but also because it replicates, in essence, the first pair of experiments with two SIENA 1.4 servers.

7.2.3. Results

The results of the SIENA experiments were mixed, further demonstrating the viability of the prototype yet identifying some of the limits of both the specification languages and the prototype itself.

The first difficulty was encountered during the process of specifying the SIENA 1.4 protocol, even prior to the first experiment. SIENA messages are allowed to be freeform, meaning that arbitrary whitespace (in the form of spaces, tabs, and line terminators) is allowed to occur between any two tokens. The specification languages had not originally been designed to permit a whitespace token to be defined, so it was thought that this token could simply be explicitly included in the definition of larger structures. For example, supposing the whitespace token was named `<WS>`, the PUB message can be defined as follows:

```
HEADER (
  (<WS>)? <SENP> (<WS>)? <OPEN> (<WS>)?
  METHOD_PUB (<WS>)? TO (<WS>)? (VERSION (<WS>)?)?
  <CLOSE>
)
PUBLICATION (HEADER (<WS>)? (EVENT | EVENTS) (<WS>)?)
  = PUBLICATION
```

However, not only does this clutter the specification, but it essentially doubles the lookahead required to differentiate SIENA messages from one another, which would then require a lookahead greater than eight (the maximum allowable by the specification languages). This left us with the option of increasing the lookahead or defining a mechanism to allow the specification of ignorable tokens. Increasing the lookahead was considered undesirable, not only because it could significantly decrease the performance of the parser but also because it does not solve the tangential problem of specification clutter.

Consequently, a mechanism to allow whitespace was developed, so that the above two productions became:

```
HEADER (<SENP> <OPEN> METHOD_PUB TO (VERSION)? <CLOSE>)
PUBLICATION $ignore(<WS>) (HEADER (EVENT | EVENTS))
  = PUBLICATION
```

The "ignore" directive in the root production, `PUBLICATION`, indicates that `<WS>` should be ignored between every pair of tokens. Further, this directive is propagated through all subproductions, so that it need not be repeatedly specified. However, some subproductions do not permit whitespace between their tokens, a restriction for which we needed to compensate. The production that defines a string value, for example, needed to be modified as follows:

```
STRING_VALUE $regard (<QUOTE> <STRING> <QUOTE>)
  = STRING_VALUE
```

The “regard” directive in this production indicates that any ignorable token production specified by a token or structure that references this production should not be matched.

The free-form messages of SIENA could now be specified, without explicitly defining every point at which whitespace could occur. A second difficulty was immediately identified, however: SIENA does not specify the order of headers, including the header that specifies the message type. This indeterminate ordering prevents us from (conveniently) specifying true SIENA messages, since our specification languages do not permit such indeterminate structures without risking an invalid message (allowing constructions such as multiple method type headers). In this case, we had the option either to further modify the specification languages or to modify the definition of SIENA messages.

Modifying the specification languages was determined to be impractical in this case, since, even were such a modification made, the type of message would not be determinable within a (reasonable) fixed lookahead. So, technically, SIENA turned out to be impossible to specify using our technique. Fortunately, the order in which SIENA servers generate the headers for any given message is fixed, despite the ambiguous specification. The version header is first, and the method header is second, making the type of message uniquely identifiable within precisely eight tokens. Although this does not conform to the technical specification of the protocol, it does conform to the implemented specification of the protocol. However, we note that, in the general case, protocols that use this specification technique are not well suited for the results of this work.

After those two issues had been settled, the specification for SIENA 1.4 was ready for testing. The first pair of experiments, which involved one SIENA 1.4 server initiating interactions with another SIENA 1.4 server, was intended to judge how well the prototype handled multiple sequential interactions using a common protocol, in addition to testing the reversed roles of the translators (i.e., translators sending concept events to one another, rather than protocol messages). Some timing data (compiled below) was also gathered, in order to determine relative slowdown when compared to two SIENA servers connected under normal circumstances (i.e., without intervening translators. These experiments completed successfully, with no difficulties beyond the initial specification troubles.

The second pair of experiments, which involved a SIENA 1.4 server initiating interactions with a SIENA 1.5 server, was intended to test the ability to match these protocols with a mandatory, deprecated interaction and to gather further timing data. As mentioned in the previous section, the addition of a filter was required in order to reflect stateful data of the interaction back to the initiator-side translator. This also identified a limitation of the technique, namely that a message reliant on application-provided data cannot be automatically composed, and consequently such a message, if required as a response during an interaction of the protocol, cannot be deprecated in a future version of the protocol for the techniques here to still be applicable.

The third pair of experiments, which involved a SIENA 1.5 server initiating interactions with a SIENA 1.4 server, was intended to validate that the problematic response in the previous pair of tests would not impact the inverse setup. These experiments also completed successfully, and the last of the timing data was gathered.

Compiled below is the timing data, from the second of each of the pairs of experiments, using the default case (no translators) as a baseline. The numbers are an average of twenty runs for each case (though one baseline run, two runs of case [2], and one run of case [3] were discarded as outliers; additional runs were performed as replacements).

Case	Startup (translators)	Startup (SIENA)	Publications	Increase
[0] (baseline)	N/A	0.47 seconds	3.62 seconds	N/A
[1] 1.4 → 1.4	4.80 seconds	0.51 seconds	16.8 seconds	464%
[2] 1.4 → 1.5	5.23 seconds	0.44 seconds	17.9 seconds	494%
[3] 1.5 → 1.4	4.96 seconds	0.39 seconds	16.6 seconds	459%

The translator start-up time includes the startup time for the filter application in case [2]. The SIENA start-up time includes the WHO/INF interaction for case [0], case [1], and case [2], and the SIENA client subscription for all cases. (No WHO/INF interaction was required in case [3].) The publications column represents the average amount of time required to send ten thousand (of the same) publications from one server to another. The last column shows the percentage increase in the time required for the publication interactions over the baseline.

Start-up costs are considered to be negligible, given that they occur only once during the run-time life of the distributed application, an application which is expected to run for a significantly longer period of time. However, the difference in combined start-up costs (approximately 1000%, double the increase in message passing cost) would make this technique impractical for a distributed application that might run only a single interaction during its run-time life.

In all three (non-baseline) cases, the increase in publication time is nearly 500%. At least a 200% increase was expected, since each message is composed and parsed twice: composed by the first SIENA server, parsed by the first translator, composed again by the second translator, and parsed again by the second SIENA server. The remaining difference is suspected to come from the overhead of ACT construction and composition (particularly since JavaCC generates relatively fast parsers). Some of the increase may also come from the number of (small) concept events being sent and processed between the translators, which also use a TCP connection for communication.

Regardless, the overhead will be linear with respect to the number of publications, and is thus judged reasonable for the SIENA application. Various improvements to the prototype may also increase performance.

7.3. Weblogs

Unlike HTTP and SIENA, WebLogs did not lend itself to specification using this technique. Although the two WebLogs interfaces (the Blogger API and the MetaWeblog API) were identified as a potential candidate for specification, they are defined by interfaces rather than protocols, using XML-RPC as the communication protocol. (Although it may be possible to hook a future version of the prototype into one or more standard Java interfaces or other language interfaces, particularly for the purpose of dynamic validation, it was impossible to do so with the current prototype barring a major redesign.) The basic WebLogs architecture is shown in Figure 73.

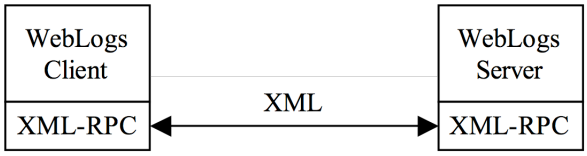

Figure 73: Simple XML-RPC architecture

The client component allows a method call as if it had a direct reference to the server. The client XML-RPC front-end then automatically wraps the call as an XML document and sends it to the server XML-RPC front-end; the server-side XML front-end then unpacks the method call and performs the call on the server component. The result of the method call is sent back in an analogous manner.

The two WebLogs APIs in question are not compatible with each other, even using a common carrier protocol, meaning that a client using one interface cannot contact a server that provides the other. The goal is to somehow unify these two interfaces, allowing some uniform communication for the WebLogs community. For that we require a specification for each message exchanged in the two API versions, one pair of messages (call and return) corresponding to each of their methods as translated by an XML-RPC application.

The question then becomes one of specifying XML-RPC. Since XML is an LL(1) language, it stands to reason that our specification limitations should still allow XML-RPC messages to be specified, though the lookahead required to differentiate messages may be significantly greater than eight symbols, particularly with the leading XML header. Still, it is theoretically possible. However, the different implementations of XML-RPC seem to have different standards: Some use `<4i>` tags to wrap an integer element and some use `<int>` tags, some use `<string>` tags

to wrap a string element and some use no tags at all (defaulting to a string value), etc. All of these and more significant variants (e.g., different headers) seem to be accepted by the lenient XML-RPC parsers.

This leaves us with the difficult task of specifying every possible combination of variants for each message corresponding to an interface method call (and return), even if they could all be known. (XML-RPC is intended as a technology to "hide" the details of communication from the developer, and consequently the precise format of XML-RPC messages is somewhat dependent on the implementation.) Given these factors, it was decided that the WebLogs case study was not feasible.

Instead, however, the techniques described here may offer a suitable replacement for XML-RPC if a more stable set of protocol messages can be determined. Although it may be possible to apply this work to this use of XML-RPC, doing so would be nontrivial and the specification of a protocol particular to WebLogs may instead be a more viable solution. For that, however, we require more precise knowledge of the inner-workings of the APIs than is currently available, particularly given that the implementation of the MetaWeblog API is proprietary, embedded within the products of UserLand Software, Inc.[17]

---

[17] http://www.userland.com/

8. Conclusions

We have presented a set techniques that accomplishes dynamic protocol evolution. These techniques are different from other methodologies that only solve a portion of the problem or require the adoption of a particular communication protocol/paradigm. Instead, a more encompassing solution is proposed that allows the use of existing communication protocols. Further, the solution can be adapted, at some performance cost, for use with existing software.

The key contributions of this work are:

1) A broader technique for event-based parsing (*viz.*, the events that can be produced by a parser are automatically derived from a specification, rather than defined by the parser).
2) An innovative approach to the construction and composition of structured messages from concept/value events.
3) The melding of these techniques with interaction-state tracking to permit the management and translation of a protocol, within a translator component.
4) The means by which such translators can be automatically generated.

5) A framework that governs the coordination of the generation and use of such translators.
6) Preliminary analysis that verifies the feasibility and tractability of the above.

Consequently, our solution involves a coordination framework, which incorporates a component that negotiates the communication protocol to use, a component that generates other components capable of translating protocol messages into concept events and *vice versa*, and the components that perform the translation. The translator components encapsulate the tasks of deconstructing messages of a specific version of a specific protocol into concept events, the inverse task of constructing messages from concept events, and the validation of sent and received message sequences that comprise an interaction.

The experience of protocol specification and prototype implementation has indicated that this work is an important first step in a more comprehensive effort for a robust system of achieving dynamic protocol evolution on a large scale. Analysis indicates constraints on the protocols to which this work is applicable, as well as several possibilities for future work.

8.1. Limitations

This work was targeted at application-level, message-oriented communication protocols. This self-imposed limitation can be relaxed somewhat, as will be discussed in the next section. However, for the purposes of the work presented in this document the limitation is explicit. Through experience, other limitations were either necessary for specification or discovered as a consequence.

Within this broad paradigm of communication, we impose further syntactic restrictions according to our choices for the definitions of the specification languages. In particular, interactions, which define sequences of messages, are considered describable by a state machine and thus a regular language. Messages are expected to conform to an LL(1) language, with the exception of context-sensitive termination. Context-sensitive termination is limited to countable repetitions that have additional restrictions, namely with respect to the prohibition of nesting instances of the concept used to represent the count. Further, there is the restriction that all messages that can be received by a role in a given interaction state must be differentiable within eight tokens. Finally, this last restriction applies to every message that, when received by a particular role, indicates a new interaction. Other features of the specification languages are absent and worth considering; these will also be covered in the next section.

Beyond the syntactic restrictions of the specification languages, protocols must have certain features in order for them to be a candidate for dynamic evolution. Each message of the protocol must have a differentiating feature that occurs within eight tokens. (Note that this is a stronger statement than mere differentiability within eight tokens.) Each “relevant” token of each message must also be uniquely identifiable by concept association; this means that two tokens cannot be perceived as representing the same concept if the application interested in those concepts must distinguish between the two. Further, the order of concepts as provided to an application must be considered “loose” in the sense that applications should not be reliant on a strict ordering of the concepts produced as a message is parsed. In fact,

order should be considered relevant only when a message could contain one or more tokens or structures that are associated with the same concept. (Unfortunately, this includes "repeatable" tokens and structures that are associated with a concept. Consequently, it is expected that almost all sequences of events produced by a parsed message can introduce potential ordering dependencies.)

Beyond such limitations are those that prevent an evolution. These include, most notably, the inclusion of a necessary token concept in a required message of a more recent protocol version, when the token associated with the concept cannot be assigned a default value; this will prevent any such message from ever being composed, unless an intermediate solution (as in the case of SIENA, for example) can be found. The inverse of this primary constraint, when an application is *required* to understand a concept event produced by a parse of a message of a more recent protocol version, is also a restriction. Certain re-orderings of concepts as produced by parsing or as received for composition can also prevent an evolution. Another significant limitation is that tokens in an earlier version cannot be promoted to structures in a later version, given that any associated concepts have different classifications; such a change would likely confuse applications. Finally, of course, every new version of a protocol must adhere to all of the previous restrictions.

Despite these limitations, a significant degree of flexibility is allowed in moving from one protocol version to another. As was demonstrated by the case studies, certain tokens can be deprecated (such as tokens that represent version numbers) and others added without significant impact, particularly if constant values or carefully chosen default values are assigned to those tokens. Structures can be

modified often without impact. (The primary use for structure concepts seems to be as a means for giving the application feedback, information that is useful only if the application is at least somewhat aware of the internal format of a message.) Even interactions can be modified, as was shown by the HTTP example, or deprecated (with restriction), as was shown by the SIENA example.

8.2. Future Work

Despite the apparent flexibility, there are several areas of potential expansion for the techniques described here. Many of these have already been mentioned, at least tangentially, in the more detailed discussions of previous chapters.

Message hierarchies. Currently, messages are kept in a global namespace and defined individual in their entirety. However, messages of a protocol often seem to have a common structure that can be exploited to avoid the LL(k) lookahead necessary to differentiate these similar messages from one another.

Concept hierarchies. In the same vein as message hierarchies, concept hierarchies could be exploited as a means to allow greater flexibility with respect to both the recognition of concepts produced by parsing and the construction of a message via individual concept events. This could have significant impacts on the algorithm for ACT construction.

Message transformation via inconsistent concept values. Currently, the value included with a token concept added to an ACT is supposed to adhere to the token specification. However, it may be possible to use concept hierarchies as a means to perform message transformations with values that would otherwise be viewed as inconsistent.

Clarification of context-sensitive termination. Several aspects of message composition make context-sensitive termination a complicated process (whereas parsing counted structures is relatively trivial). Formalizing the rules for counted compositions, particularly when they are nested or when multiple tokens may be associated with the same concept as is the "counter" token, should be a priority.

Additions to context-sensitive termination. As mentioned in Section 4.3.2, there are two different kinds of context-sensitive termination, of which we handle only one. The feasibility of the other, that of a defined terminator (as is the case with Multipart MIME), should be further evaluated.

Composition hints. Applications aware of the internal format of a message should be able to make use of this information by providing "hints" to the ACT when a concept is added. Not only would this allow an application to more accurately describe its intention in providing such a concept, but it would likely be a significant optimization in constructing messages.

Maximizing composition subtrees. Parallel composition techniques and the like can also offer an optimization to the ACT construction process, though the real strength would be in making a failed message composition less likely.

Harvesting discarded/remaining concepts provided to an ACT. The current ACT construction algorithm simple throws away unused concepts (i.e., concepts for which no associated token was found). Additionally, any concepts provided by the application and not used during message composition are lost. Under certain circumstances, it may be desirable to apply these unused concepts to the composition of additional messages (e.g., when the interaction-state tracker has

determined that another message must be sent in order to reach a state recognizable by the application). Determining the circumstances under which this should occur presents an interesting problem.

Parameterized default values. As was raised by the introduction of the "Host" header in the HTTP case study, parameterized default values would be of great assistance in some instances. The specification of a message may rely on particular, fixed values that lie somewhere in between a value that must be provided by the application and a constant default value. How these defaults should be determined, and more importantly how the specification languages should incorporate them, is a matter for further study.

Additional specification language features. In particular, the specification of connection timeouts and when connections should otherwise be closed is an important issue that should be addressed for the next version of the prototype.

Role-specific translators. Every translator generated by the prototype is capable of recognizing the protocol from the perspective of any role that may participate in the protocol. This is not necessary under most circumstances, and an optimization my be achieved by generating translators that are more tailored.

Filter intermediaries. The automatic generation of filter components that could be interposed between two translators, as in the SIENA case study, is a large problem with far-reaching implications for the set of applicable protocols.

More general perspective of protocol families. What defines a protocol as a later version of another protocol is somewhat arbitrary. The differences between two protocols that are unrelated may actually be fewer than the differences between two

versions of the “same” protocol. What actually define a protocol are the concepts encapsulated by the protocol, both in terms of the messages and the interactions. Certain concepts are clearly more important than others, and an explicit classification should be developed, so that an application known to handle a certain set of (possibly ordered) concepts can be unquestionably matched to one or more protocols whose concepts sets are compatible.

This will require rethinking the notion of how applications are matched to protocols, which is currently by name and version as a convenience.

Extension to object-oriented systems. Every object-oriented system can be viewed as implementing a protocol, as method calls and returns are isomorphic to messages (a notion made explicit by such tools as XML-RPC). Methods must be called in a prescribed order that is often not captured by classical interface specifications, and such orders may be specifiable as interactions.

The conjunction of a set of interfaces that describe the architecture of an object-oriented system and a tool similar to the prototype developed for this work may be useful for the run-time validation of the system. Event analysis and interaction-state tracking may reveal logical errors that are otherwise unfound by static methods.

Beyond such validation, the possibility of gleaning a useful evaluation of system compatibility according to the restrictions of protocol evolution is worth investigating.

BIBLIOGRAPHY

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[Alb91] H. Alblas. Introduction to Attribute Grammars. In *Attribute Grammars, Applications and Systems, SAGA Proceedings*, H. Alblas and B. Melichar, eds., Lecture Notes in Computer Science, volume 545, pages 1–15. Sprinter-Verlag, 1991.

[AP93] M. B. Abbot and L. L. Peterson. A Language-Based Approach to Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, February 1993.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools* ("The Dragon Book"). Addison-Wesley, 1996.

[Bac59] J. W Backus. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132. UNESCO, 1959.

[Bay93] Bayfront CAPE Tools User's Guide, version 1.5. Bayfront Technologies, 1993.

[BD99] J. Brunekreef and B. Diertens. Toward a User-Controlled Software Renovation Factory, In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 83–90. IEEE Computer Society, March 1999.

[BDS93] C. M. Bowman, P. B. Danzig, and M. F. Schwartz. Research Problems for Scalable Internet Resource Discovery. Technical report CU-CS-643-93, University of Colorado at Boulder, March 1993.

[Bel03] G. Bella. Availability of Protocol Goals. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 312–317. ACM Press, 2003.

[BFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hyptertext Transfer Protocol—HTTP/1.0. Internet Requests for Comment (RFC) 1945, May 1996.

[BHM+03] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture. W3C Working Group Note, February 2004.

[Boy96] J. T. Boyland. Conditional Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73–108, January 1996.

[BPS+04] T. Bray, J. Paoli, C. M Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, February 2004.

[BS91] F. Boussinot and R. de Simone. The ESTEREL Language. In *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[BST+94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca Model of Software-System Generators. *IEEE Software*, 11(5):89–94, September 1994.

[BSV00] M. van den Brand, A. Sellink, and C. Verhoef. Generation of Components for Software Renovation Factories from Context-Free Grammars. *Science of Computer Programming*, 36(2-3):209–266, March 2000.

[Bur65] W. H. Burkhardt. Universal Programming Languages and Processors: A Brief Survey and New Concepts. In Proceedings of AFIPS Fall Joint Computing Conference, pages 27:1–21, 1965.

[But97] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[BV96] M. van den Brand and E. Visser. Generation of Formatters for Context-Free Languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.

[BVM+04] R. Buckley, D. Venable, L. McIntyre, G. Parsons, and J. Rafferty. File Format for Internet Fax. Internet Requests for Comment (RFC) 2301, May 2004.

[Cam88] R. D. Cameron. An Abstract Pretty Printer. *IEEE Software*, 5(6):61–67, November 1988.

[Car98] A. Carzaniga. Architectures for an Event Notification Service Scalable to Wide-Area Networs, Ph.D. thesis, 1998.

[CBB+03] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[CCD+94] C. Castelluccia, I. Chrisment, W. Dabbous, C. Diot, C. Huitema, E. Siegel, and R. de Simone. Tailored Protocol Development Using Esterel. Technical Report 2374, INRIA, October 1994.

[CD98] C. Cracknell and A. C. Downton. Document Image Understanding of Handwritten Forms Using Rule-Trees. In *Proceedings of the Fourteenth International Conference on Pattern Recognition*, volume 1, pages 936–938. IEEE Computer Society, 1998.

[CDO97] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, 1997.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CFM+97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java Just in Time. *IEEE Micro*, 17(3):36–43, May-June 1997.

[CKM+03] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The Next Step in Web Services. *Communications of the ACM*, 46(10):29–34, October 2003.

[Cla98] C. Clark. Overlapping token definitions. *ACM SIGPLAN Notices*, 33(12):20–24, December 1998.

[CMU96] R. A. Cole, J. Mariani, H. Uszkoreit, A. Zaenen, and Victor Zue, eds. *Survey of the State of the Art in Human Language Technology*. Cambridge University Press, 1996.

[DAP+93] P. Druschel, M. B. Abbot, M. Pagels, and L. L. Peterson. Network Subsystem Design. *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4):8–17, July 1993.

[DSE98] R. Dale, D. Scott, and B. di Eugenio. Introduction to the Special Issue on Natural Language Generation. *Computational Linguistics*, 24(3):346–353, September 1998.

[EVD89] P. H. J. van Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS*, Elsevier, 1989.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. Internet Requests for Comment (RFC) 2616, January 1999.

[FKK96] A. O. Freir, P. Karlton, and P. C. Kocher. The SSL Protocol, Version 3.0. IETF Internet-Draft, November 1996.

[GJ90] D. Grune and C. J. H. Jacobs. Parsing Techniques: A Practical Guide. Ellis Horwood, New York, 1990.

[GLH+92] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM, 35(2):121–130, February 1992.

[HC98] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). Internet Requests for Comment (RFC) 2409, November 1998.

[Her03] C. Hertel. *Implementing CIFS: The Common Internet File System*, Prentice-Hall, New Jersey, 2003.

[Hoa85] C. A. R. Hoare. *Communication Sequential Processing*. Prentice-Hall, New Jersey, 1985.

[Hor51] A. Horn. On Sentences Which Are True of Direct Unions of Algebras. *Journal of Symbolic Logic*, 16:14–21, 1951.

[IEEE99] IEEE Standard for Information Technology—Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks—Specific Requirements—Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Computer Society, 1999.

[IEEE04] IEEE Standard for Information Technology—Portable Operating System Interface (POSIX)—Technical Corrigendum 2. IEEE Computer Society, 2004.

[JRW02] C. Jones, A. Romanovsky, and I. Welch. A Structured Approach to Handling On-Line Interface Upgrades. In Proceedings of the Workshop on Dependable On-Line Upgrading of Distributed Systems, pages 1000–1005. IEEE Computer Society, August 2002.

[KA98] D. Krieger and R. M. Adler. The Emergence of Distributed Component Platforms. *IEEE Computer*, 31(3):43–53, March 1998.

[KCS+00] I. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja. Techniques for Obtaining High Performance in Java Programs. *ACM Computing Surveys (CSUR)*, 32(3):213–240, September 2000.

[KHB02] J. Kulik, W. Heilzelman, H. Balakrishnan. Negotiation-Based Protocols for Disseminating Information in Wireless Sensor Networks. *Wireless Networks*, 8(2/3):169–185, 2002.

[Kle01] J. Klensin. Simple Mail Transfer Protocol. Internet Requests for Comment (RFC) 2821, April 2001.

[Knu62] D. Knuth. A History of Writing Compilers. *Computers and Automation*, December 1992.

[Knu65] D. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8(6):607–639, December 1965.

[Knu04] D. Knuth. *Selected Papers on Computer Languages*. University of Chicago Press, 2004.

[KR01] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, 2001.

[Lev97] E. Levinson. The MIME Multipart/Related Content-Type. Internet Requests for Comment (RFC) 2112, March 1997.

[LKA+95] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[LLW+04] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, eds. Document Object Model (DOM) Level 3 Core Specification, version 1.0. W3C Recommendation, April 2004.

[LMB92] J. Levine, T. Mason, and D. Brown. *Lex and YACC*, 2nd edition. O'Reilly, 1992.

[Lov77] D. B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[Luc96] D. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events. In *Proceedings of the DIMACS Partial Order Methods Workshop IV*. Princeton University, July 1996.

[Lus94] F. Lustman. Specifying Transaction-Based Information Systems with Regular Expressions. *IEEE Transactions on Software Engineering*, 20(3):207–217, March 1994.

[LW98] G. R. Lowry and A. I. Wallace. Automating the Generation of Objects. In *Proceedings of the IEEE 1998 International Conference on Education and Practice*, pages 150–157. IEEE Computer Society, January 1998.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, second edition. Addison-Wesley, 1999.

[MB02] W.S. Means and M.A. Bodie. *The Book of SAX: The Simple API for XML*. No Starch Press, 2002.

[MDE+95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference*, number 989 in Lecture Notes in Computer Science, pages 137–153. Springer-Verlag, September 1995.

[MFG+97] J. Jogul, R. Fielding, J. Gettys, and H. Frystyk. Use and Interpretation of HTTP Version Numbers. Internet Requests for Comment (RFC) 2145, May 1997.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, New York, 1982.

[MSS+98] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). Internet Requests for Comment (RFC) 2408, November 1998.

[Nei84] J. M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. IEEE Transactions on Software Engineering, SE-10(5): 564–574, September 1984.

[Par93] T. Parr. Obtaining Practical Variants of LL(k) and LR(k) for k>1 by Splitting the Atomic k-Tuple, Ph.D. dissertation, 1993.

[PC96] N. Pryce and S. Crane. Communication and Configuration in Distributed Systems. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 144–151. IEEE Computer Society, May 1996.

[PE88] F. Pfenning and C. Elliot. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[Pla03] D. S. Platt. *Introducing Microsoft .NET*, third edition. Microsoft Press, 2003.

[PR85] J. Postel and J. Reynolds. File Transfer Protocol (FTP). Internet Requests for Comment (RFC) 959, October 1985.

[Res01] P. Resnick. Internet Message Format. Internet Requests for Comment (RFC) 2822, April 2001.

[Reu03] R. H. Reussner. Automatic Component Protocol Adaption with the CoCoNut Tool Suite. *Future Generation Computer Systems*, 19(5):627–639, 2003.

[RW04] N. D. Ryan and A. L. Wolf. Using Event-Based Translation to Support Dynamic Protocol Evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 408–417. IEEE Computer Society, May 2004.

[Sch03] J. L. Schilling. The Simplest Heuristics May Be the Best in Java JIT Compilers. ACM SIGPLAN Notices, 38(2):36–46, February 2003.

[SHR+02] J. M. Sierra, J. C. Hernandez, A. Ribagorda, and N. Jayaram. Migration of Internet Security Protocols to the IPSEC Framework. In *Proceedings of the IEEE 36th Annual 2002 International Carnahan Conference on Security Technologies*, pages 134–143. IEEE Computer Society, October 2002.

[Sie00] J. Siegel. *CORBA 3 Fundamentals and Programming*, second edition. Wiley, 2000.

[Sim95] C. Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. Technical Report MSR-TR-95-52, Microsoft Research, 1995.

[Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1997.

[Str97] B. Stroustrup. *The C++ Programming Language*, third edition. Addison-Wesley, 1997.

[TC01] J. Tremblay and G. A Cheston. *Data Structures and Software Development in an Object Oriented Domain: Eiffel Edition*. Prentice Hall, 2001.

[THZ+03] M. Törö, Thong Tri Huynh, Jinsong Zhu, Kangming Liu, and Victor. C. M. Leung. CORBA Based Design and Implementation of Universal Personal Computing. *Mobile Networks and Applications*, 8(1):75–86, February 2003.

[TIA97] TIA Standard TAI-232-F: Interface Between Data Terminal Equipment and Data Curcit-Terminating Equipment by Employing Serial Binary Data Interchange. Telecommunications Industry Association, 1997.

[TMA+96] R. N. Taylor, N. Medvidovic, K. M. Anderson, J. E. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, 22(6):390–406, June 1996.

[Wie92] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, March 1992.

[Woo99] L. Wood. Programming Marked-Up Documents. *Markup Languages: Theory and Practice*, 1(1): 91–100, December 1999.

[WS96] D. Wagner and B. Schneier. Analysis of the SSL 3.0 Protocol. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 29–40. USENIX Press, November 1996.

[XGB02] Kaixin Xu, M. Gerla, and Sang Bae. How Effective Is the IEEE 802.11 RTS/CTS Handshake in Ad Hoc Networks. In *Proceedings of the IEEE Global Telecommunications Conference*, volume 1, pages 72–76. IEEE Computer Society, November 2002.

[YL04] T. Ylonen, C. Lonvick. SSH Connection Protocol. IETF Internet-Draft, June 2004.

Appendix A — Glossary

For the purposes of rigor and reference, this appendix contains an alphabetical list of some of the terminology used throughout this document. Italicized terms within the definitions are themselves entries in the glossary.

abstract composition tree (ACT)

A syntax tree that undergoes *latent construction* in order to be used for the legal *composition* of a *message*.

alternate

A set of sequential elements (including *rules*) for the formulation of a *message*.

application (also, ***application component***)

An application with distributed components that coordinate via message passing. Also, a component of such an application.

aware application

An *application* that can explicitly interface with an event-based *translator*.

character

An 8-bit value. Each textual character explicitly used within the specifications of this document is considered to be associated with the corresponding 8-bit ASCII value.

character set

A (possibly disjoint) collection of one or more *characters*. The use of a character set within a *protocol specification* represents exactly one of the characters in the set.

communication (also, ***interaction communication***)

The passing of a *message* from one *role* to another during an *interaction*; a communication is legitimate only when both the sending role and the receiving role are in an appropriate *state*. An interaction is partially defined as a collection of one or more well-ordered communications.

complete (ACT subtree)

The state of being able to participate in the legal *composition* of a *message*. The conditions for an ACT subtree being complete are discussed in Section 5.3.8.

composing

The action of constructing a protocol *message* from *events* according to *protocol specification*. Composing employs the actions of *latent construction* and *message composition*.

composition (also, ***message composition***)

The action of formulating a legal *message* from an *ACT*.

concept

A name used within a *protocol specification*. Each concept is considered to have some associated, agreed-upon semantic.

There are five different lowest-level classes of concept: *token* concepts, *structure* concepts, *message* concepts, *role* concepts, and *state* concepts.

construction (also, ***dynamic construction***, ***latent construction***)

The action of creating an *ACT* from *concepts* and *literal values* provided by an *aware application*.

event

An encapsulation of a *concept* as passed between a *translator* and an *aware application*.

generation (also, ***translator generation***)

The action of automatically creating a *translator* according to a *protocol specification*.

generator

A component that performs *generation*.

grammar

A concrete definition of a *language* (i.e., the instantiation of the *syntax* of a language). See also: *protocol grammar*.

initialization (also, ***interaction initialization***)

An assignment of a *state* to a *role* at the start of an *interaction*. An interaction is partially defined as a collection of one or more initializations.

interaction

A "use" of a protocol by one or more *roles*. An interaction is defined as a collection of *initializations* and well-ordered *communications*.

The specification of an interaction amounts to a DFA (deterministic finite automata), with each sent and received *message* defining a *transition* between two states of the DFA. Each role has its own (possibly partial) perspective of the DFA, however, and consequently each state of the DFA may be simultaneously labeled with several different *states*, possibly one for each role.

interaction state tracking

The action of recording the *transitions* of an *interaction*.

language

A well-defined collection of strings. (Note that, in this context, "string" is different from "literal string".)

literal (also, ***literal value***)

A *character set* or a *string*.

message (in some contexts, ***message production***)

A subclass of *nonterminal*, representing a root production of a *protocol grammar*. A message may (or may not) be associated with a message *concept*. Each message is conceptually a "document" passed (as a meaningful entity of a protocol) from one *role* to another. The sending or receipt of a message drives the *state* of the *interaction* of which the message is a part.

negotiation (also, ***protocol negotiation***)

The action of determining which *protocol* two applications will use for message passing.

negotiator

A component that performs *negotiation*.

nonterminal

A name used within a *protocol specification*. Each nonterminal has an associated *nonterminal production* as its definition, which is the composition of tokens and structures; a nonterminal therefore carries relational information in contrast to the basic information of *terminals*.

nonterminal production

A subclass of *production*, and the definition of a *nonterminal*. Each nonterminal production incorporates references to *tokens*, *structures*, or both, and is limited by LL(1) expressiveness. Note that a token referenced by a nonterminal production is necessarily a top-level token (though it may also be used as a *subtoken*).

optional (rule)

A *rule* whose *alternates* can be present in the formulation of a *message* zero times.

parsing

The action of deconstructing a protocol *message* into *events* according to *protocol specification*.

primary rule

The defining *rule* of a *production*.

production

A BNF-style definition for a terminal or nonterminal of a protocol specification.

There are three different final classes of production: *token* productions, *structure* productions, and *message* productions.

protocol

A standardized description of application-level message passing for distributed communication. A protocol is comprised of a set of *messages* and a set of *interactions* that define the order in which the message can be passed.

protocol grammar

A concrete definition of a *protocol* (i.e., the instantiation of the *syntax* of a protocol). A protocol grammar is a collection of *grammars*, one for each *message* of the protocol, in addition to the grammar (concrete definition) of each *interaction* of the protocol.

protocol specification

A set of files (or other medium) that records a *protocol grammar* and the *concepts* to which elements (*productions*) of the grammar are related or with which elements (*interactions*) of the grammar are defined.

protocol syntax

The abstract definition of a *protocol*, which includes not only the *syntax* of each *message* of the protocol, but the syntax (abstract definition) of each *interaction* of the protocol.

recognizer

A software component capable of dynamically validating the use of a *protocol* as conforming to the corresponding *protocol syntax*.

repeatable (rule)

A *rule* whose *alternates* can be present in the formulation of a *message* two or more times.

role (also, ***interaction role***)

A software component that represents a conceptual entity using a *protocol*.

rule

A set of *alternates* for the formulation of a *message*. A rule can be repeated a specified number of times.

state (also, ***interaction state***)

A conceptual state of an *interaction*. Since an interaction can be defined by a DFA, an interaction state is merely a state of the state machine that defines the interaction, as viewed from the perspective of a *role*.

string

A sequence of one or more *characters*.

structure (in some contexts, ***structure production***)

A subclass of *nonterminal*, representing a non-root, non-leaf production of a *protocol grammar*. The difference between a structure and a *token* is subtle, particularly given the notion of *subtokens* (essentially allowing a token to be a non-leaf productions); the developer is encouraged to use either, as logical and convenient.

subtoken

A *token*, a reference to which is incorporated in a *terminal production*.

syntax

The abstract definition of a *language*.

terminal

A name used within a *protocol specification*. Each terminal has an associated *terminal production* as its definition, which is the composition of literals and tokens; a terminal therefore carries basic information in contrast to the relational information of *nonterminals*.

terminal production

A subclass of *production*, and the definition of a *terminal*. Each terminal production incorporates references to literals, tokens, or both, and is limited to regular expressiveness. Note that a token referenced by a terminal production is necessarily a *subtoken* (though it may also be used as a top-level token).

token (in some contexts, ***token production***; also, ***top-level token***)

A subclass of *terminal*, representing a leaf production of a *protocol grammar*. The difference between a token and a *structure* is subtle, particularly given the notion of *subtokens* (essentially allowing a token to be a non-leaf productions); the developer is encouraged to use either, as logical and convenient.

Tokens are at the lowest level of a protocol specification and, as expected, capture the basic informational "pieces" of each *message*. A token is a meaningful unit of a *protocol grammar* insofar as smaller units (unless they are themselves tokens, i.e., subtokens) carry no information outside their context.

transition (also, ***interaction transition***)

A change in the state of an *interaction* due to a *communication*.

translation

The coordination of *composing*, *parsing*, and *interaction state tracking*.

translator

A component that performs *translation*.

Appendix B — Protocol Specification Languages

The multipart approach to protocol specification requires a number of different languages, one for each part. This appendix will present in detail the ten languages defined for the purpose of this work. First, the five concept specification languages will be given, followed by the five grammar specification languages. A reference subsection for the terminals used in the definition of these ten languages follows. The section is concluded with a brief discussion regarding the storage of specifications written using the languages defined here.

B.1. Grammar Specification Languages

A grammar specification language allows some aspect of the syntax of a protocol syntax to be defined. There are five grammar specification languages, one each for the overall protocol (i.e., the collection of interactions that defines the protocol as a whole), interactions, message productions, structure productions, and token productions.

B.1.1. Protocol Specification Language

```
start              -> interactionImports end
interactionImports -> INTERACTIONS_TAG
                      interactionImport+
interactionImport  -> GROUP_NAME
end                -> END_TAG
```

In addition to the syntactic rules defined by the grammar, one semantic rule must be enforced by an implementing system: For interactions I_1 and I_2, messages M_1 and M_2, role R, and states S_1 and S_2, if, from the perspective of R, the state of I_1 is initialized to S_1 and the state of I_2 is initialized to S_2, I_1 defines the communication of M_1 such that S_1 is listed as a prestate for R as the source (target), and I_2 defines the communication of M_2 such that S_2 is listed as a prestate for R as the source (target), then M_1 and M_2 must not refer to the same message. This second condition is critical, ensuring the determinism of the union of the state machines that describe the interactions; if violated, the protocol specification as a whole should be considered invalid.

B.1.2. Interaction Grammar Specification Language

```
start           -> messageImports
                   roleImports stateImports
                   initializations communications
                   end
messageImports  -> MESSAGES_TAG messageImport+
messageImport   -> GROUP_NAME
roleImports     -> ROLES_TAG roleImport+
roleImport      -> RUBRIC VERSION GROUP_NAME
                   AS_ALIAS GROUP_ALIAS
stateImports    -> STATES_TAG stateImport+
stateImport     -> RUBRIC VERSION GROUP_NAME
                   AS_ALIAS GROUP_ALIAS
transition      -> CONCEPT_NAME (OP_ALIAS GROUP_ALIAS)?
                   CONCEPT_NAME (OP_ALIAS GROUP_ALIAS)?
initializations -> INITIALIZATIONS_TAG initialization+
initialization  -> transition
communications  -> COMMUNICATIONS_TAG communication+
communication   -> PRODUCTION_NAME
                   OP_SOURCE prestates transition
                   OP_TARGET prestates transition
prestates       -> QB_PRESTATES
                   CONCEPT_NAME (OP_ALIAS GROUP_ALIAS)?
                   (
                    DL_PRESTATE
                    CONCEPT_NAME (OP_ALIAS GROUP_ALIAS)?
                   )*
                   QE_PRESTATES
end             -> END_TAG
```

In addition to the syntactic rules defined by the grammar, several semantic rules must be enforced by an implementing system:

1) The interaction must be initialized once from the perspective of each role potentially participating in the interaction. This condition is critical; if violated, the protocol specification as a whole should be considered invalid.
2) The interaction should not be initialized more than once from the perspective of each role potentially participating in the interaction; only the last such initialization should be used for the definition of the interaction.
3) No collection of prestates should list a state more than once (even via different aliases, e.g., `S@ALIAS1`, `S@ALIAS2`); superfluous prestates should be ignored.
4) For message *M*, role *R*, and state *S*, no two communications that send *M* should exist such that, for both communications, *S* is listed as a prestate from the perspective of *R* and *R* is either the source or the target (or both) in both instances; only the last such communication for the triplet of *M*, *R*, and *S* should be used for the definition of the interaction. (This condition ensures the determinism of the state machine that describes an interaction.)

For example, suppose we are given the following interaction specification, using the language defined above:

```
[messages]
  M1 M2
[roles]
  example 1.0 roles   // defines A, B
[states]
  example 1.0 states  // defines A0-A2 B0-B2
[initializations]
  A A0
  A A1
[communications]
  M1: {A0, A0} A A1 -> {B0} B B1
  M1: {A0, A1} A A1 -> {B1} B B2
  M2: {B0, B1} B B2 -> {A1} A A2
[end]
```

A warning should be issued by an implementing system for the duplicate prestate listed for the first communication of message `M1` from role `A` to role `B`. Another warning should be issued for the duplicate initialization of the interaction (once to state `A0` and once to state `A1`) from the perspective of role `A`; only the last such initialization should be used. A warning should also be issued for the communication of message `M1` from role `A` to role `B` in two different contexts (both of which list `A0` as a prestate of the interaction from the perspective of role `A`); only the last such context should be used. Finally, a warning should be issued regarding the missing initialization for role `B`.

Note that other conditions can be detected and enforced (e.g., reachability of interaction states from the perspective of each role), though an implementing system should passively allow the violation of such.

B.1.3. Message Grammar Specification Language

```
start            -> tokenImports? structureImports?
                    conceptImports?
                    definitions
                    end
tokenImports     -> TOKENS_TAG tokenImport+
tokenImport      -> RUBRIC VERSION GROUP_NAME
                    AS_ALIAS GROUP_ALIAS
structureImports -> TOKENS_TAG structureImport+
structureImport  -> RUBRIC VERSION GROUP_NAME
                    AS_ALIAS GROUP_ALIAS
```

```
conceptImports   -> CONCEPTS_TAG conceptImport+
conceptImport    -> RUBRIC VERSION GROUP_NAME
                    AS_ALIAS GROUP_ALIAS
definitions      -> PRODUCTIONS_TAG definition+
definition       -> PRODUCTION_NAME directives rule
                    (
                     AS_CONCEPT
                     CONCEPT_NAME
                     (OP_ALIAS GROUP_ALIAS)?
                    )?
directives       -> DIRECTIVE_REGARD? |
                    (
                     DIRECTIVE_IGNORE
                     QB_DIRECTIVE
                     (
                      QB_TOKEN
                      PRODUCTION_NAME
                      (OP_ALIAS GROUP_ALIAS)?
                      QE_TOKEN
                     )?
                     QE_DIRECTIVE
                    )?
rule             -> QB_RULE
                    choice (DL_RULE_CHOICE choice)*
                    QE_RULE
                    (pattern | counter)?
pattern          -> OP_RULE_0OR1 |
                    OP_RULE_0PLUS |
                    OP_RULE_1PLUS
counter          -> QB_COUNTER QB_TOKEN
                    PRODUCTION_NAME
                    (OP_ALIAS GROUP_ALIAS)?
                    QE_TOKEN QE_COUNTER
choice           -> alternate
alternate        -> (sequence directives)+
sequence         -> token | structure | rule
token            -> QB_TOKEN
                    OP_INTERACTION_LABEL?
                    PRODUCTION_NAME
                    (OP_ALIAS GROUP_ALIAS)?
                    QE_TOKEN
structure        -> PRODUCTION_NAME
                    (OP_ALIAS GROUP_ALIAS)?
end              -> END_TAG
```

In addition to the syntactic rules defined by the above grammar, several semantic rules must be enforced by an implementing system:

1) One production must be named the same as the group; this production is the message production. This condition is critical; if violated, the protocol specification as a whole should be considered invalid.

2) No two named productions should have the same name; if so, only the last such production should be used.
3) No production other than the message production should be associated with a message concept; if so, the association should be ignored.
4) No production other than the message production should include directives external to the primary rule of the definition of the production; if so, the directives should be ignored.

Note that other conditions can be detected and enforced (e.g., unused message-specific structure productions), though an implementing system should passively allow the violation of such.

Note also that the language allows the definitions of messages and structures that are (much) more powerful than LL(1). The condition of LL(1) must be enforced (with the exception of context-sensitive repetition); if this condition is violated, the protocol specification as a whole should be considered invalid.

B.1.4. Structure Grammar Specification Language

```
start             -> tokenImports? structureImports?
                     conceptImports?
                     definitions
                     end
tokenImports      -> TOKENS_TAG tokenImport+
tokenImport       -> RUBRIC VERSION GROUP_NAME
                     AS_ALIAS GROUP_ALIAS
structureImports -> TOKENS_TAG structureImport+
structureImport   -> RUBRIC VERSION GROUP_NAME
                     AS_ALIAS GROUP_ALIAS
conceptImports    -> CONCEPTS_TAG conceptImport+
conceptImport     -> RUBRIC VERSION GROUP_NAME
                     AS_ALIAS GROUP_ALIAS
definitions       -> PRODUCTIONS_TAG definition+
definition        -> PRODUCTION_NAME rule
                     (
                      AS_CONCEPT
                      CONCEPT_NAME
                      (OP_ALIAS GROUP_ALIAS)?
                     )?
```

```
directives       -> DIRECTIVE_REGARD?
                    (
                     DIRECTIVE_IGNORE
                     QB_DIRECTIVE
                     (
                      QB_TOKEN
                      PRODUCTION_NAME
                      (OP_ALIAS GROUP_ALIAS)?
                      QE_TOKEN
                     )?
                     QE_DIRECTIVE
                    )?
rule             -> QB_RULE
                    choice (DL_RULE_CHOICE choice)*
                    QE_RULE
                    (pattern | counter)?
pattern          -> OP_RULE_0OR1 |
                    OP_RULE_0PLUS |
                    OP_RULE_1PLUS
counter          -> QB_COUNTER QB_TOKEN
                    PRODUCTION_NAME
                    (OP_ALIAS GROUP_ALIAS)?
                    QE_TOKEN QE_COUNTER
choice           -> alternate
alternate        -> (sequence directives)+
sequence         -> token | structure | rule
token            -> QB_TOKEN
                    OP_INTERACTION_LABEL?
                    PRODUCTION_NAME
                    (OP_ALIAS GROUP_ALIAS)?
                    QE_TOKEN
structure        -> PRODUCTION_NAME
                    (OP_ALIAS GROUP_ALIAS)?
end              -> END_TAG
```

In addition to the syntactic rules defined by the grammar, one semantic rule must be enforced by an implementing system: No two productions should have the same name; if so, only the last such production should be used.

Note that the language allows the definitions of structures that are (much) more powerful than LL(1). The condition of LL(1) must be enforced (with the exception of context-sensitive repetition); if this condition is violated, the protocol specification as a whole should be considered invalid.

B.1.5. Token Grammar Specification Language

```
start            -> tokenImports?
                    conceptImports?
                    definitions
                    end
tokenImports     -> TOKENS_TAG tokenImport+
tokenImport      -> RUBRIC VERSION GROUP_NAME
                    AS_ALIAS GROUP_ALIAS
conceptImports   -> CONCEPTS_TAG conceptImport+
conceptImport    -> RUBRIC VERSION GROUP_NAME
                    AS_ALIAS GROUP_ALIAS
definitions      -> PRODUCTIONS_TAG definition+
definition       -> PRODUCTION_NAME directives rule
                    (
                     AS_CONCEPT
                     CONCEPT_NAME
                     (OP_ALIAS GROUP_ALIAS)?
                    )?
directives       -> DIRECTIVE_NONE? |
                    (
                     DIRECTIVE_FORCIBLE?
                     DIRECTIVE_OPTIONAL
                    )?
                    DIRECTIVE_CONSTANT?
rule             -> QB_RULE choice
                    (DL_RULE_CHOICE choice)*
                    QE_RULE
                    (pattern | counter)?
pattern          -> OP_RULE_0OR1 |
                    OP_RULE_0PLUS |
                    OP_RULE_1PLUS
counter          -> QB_COUNTER QB_TOKEN
                    PRODUCTION_NAME
                    (OP_ALIAS GROUP_ALIAS)?
                    QE_TOKEN QE_COUNTER
choice           -> alternate
alternate        -> sequence+
sequence         -> charset | string | token | rule
charset          -> OP_CHARSET_COMPLEMENT?
                    QB_CHARSET
                    charsetComponent
                    (
                     DL_CHARSET_COMPONENT
                     charsetComponent
                    )*
                    QE_CHARSET
charsetComponent -> character
                    (OP_CHARSET_RANGE character)?
character        -> QB_CHARACTER CHARACTER QE_CHARACTER
string           -> QB_STRING STRING QE_STRING
token            -> QB_TOKEN
                    OP_INTERACTION_LABEL?
                    PRODUCTION_NAME
                    (OP_ALIAS GROUP_ALIAS)?
                    QE_TOKEN
end              -> END_TAG
```

In addition to the syntactic rules defined by the grammar, one semantic rule must be enforced by an implementing system: No two productions should have the same name; if so, only the last such production should be used.

B.2. Concept Specification Languages

A concept specification language allows some class of concepts (to be associated with some element of a protocol's syntax) to be defined. There are five concept specification languages, one each for role concepts (more precisely, interaction role concepts), state concepts (more precisely, interaction state concepts), message concepts, structure concepts, and token concepts. However, each of these languages is used merely to list a set of names, each of which (by itself) defines a concept; therefore, the specification of all five languages is identical.

```
start       -> definitions end
definitions -> CONCEPTS_TAG definition+
definition  -> CONCEPT_NAME
end         -> END_TAG
```

In addition to the syntactic rules defined by the grammar, one semantic rule must be enforced by an implementing system: No two concepts should have the same name; if so, only the last such concept should be used.

B.3. Token Productions

A production for each terminal (and subterminal) used by at least one of the specification languages is listed below, divided into the following eleven categories: characters, character sets, quotes, delimiters, assignment operators, other operators, elements, directives, tags, whitespace, and comments. Note that most of the terminals are ultimately used by the nonterminal productions of more than one language.

B.3.1. Characters

Named character productions define special characters that find individual use as a terminal or subterminal (typically the latter), and are so named for specification readability. Note that the line terminator production, `CRLF`, is considered (for the purpose of convenience) to be a special named character.

```
CH_TAB           -> "\t"
CH_LF            -> "\n"
CH_CR            -> "\r"
CH_SPACE         -> " "
CH_QUOTE         -> "\""
CH_DOLLAR        -> "$"
CH_AMPERSAND     -> "&"
CH_PAREN_OPEN    -> "("
CH_PAREN_CLOSE   -> ")"
CH_ASTERISK      -> "*"
CH_PLUS          -> "+"
CH_COMMA         -> ","
CH_HYPHEN        -> "-"
CH_PERIOD        -> "."
CH_COLON         -> ":"
CH_LESS_THAN     -> "<"
CH_EQUALS        -> "="
CH_GREATER_THAN  -> ">"
CH_QUESTION      -> "?"
CH_AT            -> "@"
CH_BRACKET_OPEN  -> "["
CH_BRACKET_CLOSE -> "]"
CH_UNDERSCORE    -> "_"
CH_BRACE_OPEN    -> "{"
CH_BAR           -> "|"
CH_BRACE_CLOSE   -> "}"
CH_TILDE         -> "~"
CRLF             -> CH_CR? CH_LF
```

B.3.2. Character Sets

Character set productions define special collections of characters that find individual use as a terminal or subterminal (typically the latter), and are so named for specification readability. Note the multi-character escape sequences defined by the productions `CS_ESCAPE_OCTAL` (octal escapes), `CS_ESCAPE_CONTROL` (`\b`, `\t`, `\n`, `\f`, and `\r`), and `CS_ESCAPE_META` (`\"` and `\\`), each of which represents an escaped character used either individually or within a string. Octal

escape sequences can represent any 8-bit character; the special two-character escape sequences instead represent the control characters “backspace”, “tab”, “line feed”, “formfeed”, and “carriage return”, and the characters “double quote” and “backslash”, respectively.

```
CS_LETTER          -> [A-Z] | [a-z]
CS_DIGIT_ZERO      -> "0"
CS_DIGIT_NONZERO   -> [1-9]
CS_DIGIT           -> DIGIT_ZERO | DIGIT_NONZERO
CS_ID_FIRST        -> CS_LETTER
CS_ID              -> CS_LETTER | CS_DIGIT |
                      CH_UNDERSCORE
CS_OCTAL_FIRST     -> [0-3]
CS_OCTAL           -> [0-7]
CS_ESCAPE_OCTAL    -> "\\"
                      (CS_OCTAL_FIRST? CS_OCTAL)?
                      CS_OCTAL
CS_ESCAPE_CONTROL -> "\\" ("b" | "t" | "n" | "f" | "r")
CS_ESCAPE_META     -> "\\" ("\"" | "\\")
CS_ESCAPE          -> CS_ESCAPE_OCTAL |
                      CS_ESCAPE_CONTROL |
                      CS_ESCAPE_META
CS_ALL             -> "!" | [#-\[] | [\]-~] |
                      CH_SPACE | CS_ESCAPE
CS_COMMENT_C       -> [!-)] | [+-.] | [0-~] |
                      CH_SPACE | CH_TAB | CRLF
CS_COMMENT_CPP     -> [!-~] | CH_SPACE | CH_TAB
CS_COMMENT_SH      -> [!-~] | CH_SPACE | CH_TAB
```

B.3.3. Quotes

Quoting productions are paired (begin and end); each pair is used to quote another production (or collection of productions).

```
QB_TAG        -> CH_BRACKET_OPEN
QE_TAG        -> CH_BRACKET_CLOSE
QB_RULE       -> CH_PAREN_OPEN
QE_RULE       -> CH_PAREN_CLOSE
QB_COUNTER    -> CH_BRACE_OPEN
QE_COUNTER    -> CH_BRACE_CLOSE
QB_DIRECTIVE -> CH_PAREN_OPEN
QE_DIRECTIVE -> CH_PAREN_CLOSE
QB_CHARSET    -> CH_BRACKET_OPEN
QE_CHARSET    -> CH_BRACKET_CLOSE
QB_CHARACTER -> CH_QUOTE
QE_CHARACTER -> CH_QUOTE
QB_STRING     -> CH_QUOTE
QE_STRING     -> CH_QUOTE
QB_TOKEN      -> CH_LESS_THAN
QE_TOKEN      -> CH_GREATER_THAN
```

```
QB_PRESTATES -> CH_BRACE_OPEN
QE_PRESTATES -> CH_BRACE_CLOSE
```

B.3.4. Delimiters

Delimiter productions are used to delimit the components of a list of productions (or collections of productions).

```
DL_RUBRIC_COMPONENT  -> CH_PERIOD
DL_VERSION_COMPONENT -> CH_PERIOD
DL_RULE_CHOICE       -> CH_BAR
DL_CHARSET_COMPONENT -> CH_COMMA
DL_PRESTATE          -> CH_COMMA
```

B.3.5. Assignment Operators

Assignment operator productions represent some operation of assignment of a production (or collection of productions) to another production (or collection of productions).

```
AS_ALIAS   -> CH_EQUALS
AS_CONCEPT -> CH_EQUALS
```

B.3.6. Other Operators

Operator token productions represent some operation (other than assignment) or some other special denotation.

```
OP_DIRECTIVE          -> CH_DOLLAR
OP_RULE_0OR1          -> CH_QUESTION
OP_RULE_0PLUS         -> CH_ASTERISK
OP_RULE_1PLUS         -> CH_PLUS
OP_CHARSET_COMPLEMENT -> CH_TILDE
OP_CHARSET_RANGE      -> CH_HYPHEN
OP_ALIAS              -> CH_AT
OP_INTERACTION_LABEL  -> CH_AMPERSAND
OP_SOURCE             -> CH_COLON
OP_TARGET             -> CH_HYPHEN CH_GREATER_THAN
```

B.3.7. Elements

Element productions define the names (or literal values) of the primary "pieces" of a specification. Of particular interest are the `CHARACTER`, `STRING`, `RUBRIC`, `VERSION`, `GROUP_NAME`, `GROUP_ALIAS`, `CONCEPT_NAME`, and

PRODUCTION_NAME productions, which define the major elements. Note that, in contrast to C/C++/Java, the production of an identifier (ID) prohibits an underscore as an initial identifier character (as well as prohibiting the dollar sign as an identifier character, permissible in Java [cite]).

```
ID                          -> CS_ID_FIRST (CS_ID)*
INTEGER                     -> CS_DIGIT_ZERO |
                               CS_DIGIT_NONZERO CS_DIGIT*
LETTER                      -> CS_LETTER
CHARACTER                   -> CS_ALL
STRING                      -> CS_ALL+
RUBRIC_COMPONENT_NAME       -> ID
RUBRIC_COMPONENT            -> RUBRIC_COMPONENT_NAME
RUBRIC                      -> RUBRIC_COMPONENT
                               (
                                DL_RUBRIC_COMPONENT
                                RUBRIC_COMPONENT
                               )*
VERSION_COMPONENT_NUMBER -> INTEGER
VERSION_COMPONENT_LETTER -> LETTER
VERSION_COMPONENT           -> VERSION_COMPONENT_NUMBER
                               VERSION_COMPONENT_LETTER?
VERSION                     -> VERSION_COMPONENT
                               (
                                DL_VERSION_COMPONENT
                                VERSION_COMPONENT
                               )*
GROUP_NAME                  -> ID
GROUP_ALIAS                 -> ID
CONCEPT_NAME                -> ID
PRODUCTION_NAME             -> ID
```

B.3.8. Directives

Directive productions define the various directives that may be given at the start of (or within) a production definition (of a specification).

```
DIRECTIVE_NONE     -> OP_DIRECTIVE "none"
DIRECTIVE_FORCIBLE -> OP_DIRECTIVE "forcible"
DIRECTIVE_OPTIONAL -> OP_DIRECTIVE "optional"
DIRECTIVE_CONSTANT -> OP_DIRECTIVE "constant"
DIRECTIVE_REGARD   -> OP_DIRECTIVE "regard"
DIRECTIVE_IGNORE   -> OP_DIRECTIVE "ignore"
```

B.3.9. Tags

Tag productions demarcate the sections of a specification.

```
TOKENS_TAG           -> QB_TAG "tokens" QE_TAG
STRUCTURES_TAG       -> QB_TAG "structures" QE_TAG
MESSAGES_TAG         -> QB_TAG "messages" QE_TAG
ROLES_TAG            -> QB_TAG "roles" QE_TAG
STATES_TAG           -> QB_TAG "states" QE_TAG
INTERACTIONS_TAG     -> QB_TAG "interactions" QE_TAG
CONCEPTS_TAG         -> QB_TAG "concepts" QE_TAG
PRODUCTIONS_TAG      -> QB_TAG "productions" QE_TAG
INITIALIZATIONS_TAG -> QB_TAG "initializations" QE_TAG
COMMUNICATIONS_TAG  -> QB_TAG "communications" QE_TAG
END_TAG              -> QB_TAG "end" QE_TAG
```

B.3.10. Whitespace

The whitespace production is not explicitly used within the definitions of the other productions that define the specification languages, but rather defines what is considered whitespace within a free-form specification conforming to one of the specification languages.

Whitespace may be used anywhere within a specification, except immediately after a `QB_CHARACTER`, `QB_STRING`, `CHARACTER`, or `STRING` production.

```
WS -> (CH_SPACE | CH_TAB | CRLF)+
```

B.3.11. Comments

Like whitespace, comment productions are not explicitly used within the definitions of the other productions that define the specification languages, but are instead ignored in between other elements of a specification. Anywhere that whitespace may be used, comments may also be used.

As a convenience to the developer, comments may be (possibly mixed) in the form of C, C++, or shell-style comments, as preferred. Recall that C++ comments (specifically, those comments beginning with a double slash) and shell-style comments reach until the end of the current line.

```
COMMENT_C   -> "/*/" |
               "/*"
               (CS_COMMENT_C | "*"+ CS_COMMENT_C | "/")*
               "*"+ "/"
COMMENT_CPP -> "//" (CS_COMMENT_CPP)* CRLF
COMMENT_SH  -> "#" (CS_COMMENT_SH)* CRLF
```

B.4. Specification Storage

A protocol can thus be fully specified (piecemeal) by our ten specification languages. However, the manner in which the specifications are stored is left as an implementation detail, with the caveat that a qualified protocol unit (a pairing of the name and version of the relevant protocol) uniquely determines a specification for the purposes of downloading or translator generation.

One possibility naturally suggests itself: Each specification can be placed in its own file, following a directory structure that reflects a reversed Internet domain name, similar to a Java package. Grammar specifications can be maintained a "grammars" relative root directory, whereas concept specifications can be maintained in a peer "concepts" relative root directory; both of these relative root directories can then be directly maintained in a root specifications directory. In this scenario, the different specification types must be uniquely identified, even if groups of different types and of same name reside in the same subdirectory.

For example, the HTTP 1.0 specification has the following file layout in the prototype system (assuming "`.`" is the root specifications directory):

```
./
  grammars/
    org/
      w3c/
        http/
          _1_0/
            top
            itr.GET
            msg.REQUEST_GET
            msg.RESPONSE_200
            stc.header
            tkn.header
            tkn.body
            tkn.base
        uri/
          _1_0/
            tkn.base
  concepts/
    org/
      w3c/
        http/
          _1_0/
            rol.base
            cse.base
            msg.base
            tkn.header
            tkn.body
            tkn.base
        uri/
          _1_0/
            tkn.base
```

Here, the HTTP 1.0 specification as a whole is identified by the file ./grammars/org/w3c/http/_1_0/top. Other specifications are stored in files with appropriately prefixed names; for example, each interaction (in this case only one) is specified in a file with a name prefixed by "itr.". The following is a list of the prefixes and their corresponding specification:

`itr.` Interaction grammar specification
`msg.` Message grammar/concept specification
`stc.` Structure grammar/concept specification
`tkn.` Token grammar/concept specification
`rol.` Role concept specification
`cse.` State concept specification

Note the URI specification included in the file hierarchy above. While a URI is not itself a protocol, URIs are used by the HTTP protocol (and other protocols) and

consequently must have a specification that can be referenced by a specification of HTTP. The fragmentary approach to protocol specification thus also allows the conceptual separation of specifications (either grammar or concept) that are not themselves at the level of a protocol but are nevertheless required for protocol specification, without requiring a separate mechanism. Clearly, this has implications for abstraction and reuse of specification fragments.

Appendix C — ACT Algorithms

The process of composing messages automatically and deterministically from a grammar is considered a contribution of this work. The process is performed in two phases: the construction of an abstract syntax tree (ACT), and the composition of a message from a (partially) constructed tree.

The latter phase relies heavily on the first and indeed incorporates part of its algorithm. While the latter phase is intuitive and conducive to verbal description, the former phase is non-intuitive and will thus be described by a more formal notation.

C.1. Construction

The algorithm for the construction of an ACT is complex, and will thus be presented as a set of three primary processes. (For consistency throughout the algorithm, all variables are considered global.) Note that this algorithm is intended to be a rigorous evaluation of the overall process, and not necessarily a guideline for implementation.

MAIN:

1. Determine the set of messages Z such that for each message M, M is a member of Z if and only if M can be sent by role R during interaction I in state S.
2. Construct ACT T:
    1. Construct ACT root node Q of T.
    2. For each message M such that M is a member of Z:
    3. Construct ACT alternate node N.
        1. Construct placeholder node $Q^{(1)}$ that references M.
        2. Add $Q^{(1)}$ as a child node of N.
    4. Add N as a child node of Q. (Note that Q will have no child nodes if Z is empty.)
3. For each token concept C and (possibly null) literal value L provided by application A implementing R:
    1. Pass (C, L, X) to Q.
    2. Initialize the value V_Q to be returned by Q to *LACK*.
    3. For each ACT alternate node N of Q:
        1. Pass (C, L, X) to N.
        2. Initialize the value V_N to *LACK*.
        3. Initialize J to 1.
        4. For the sequence node $Q^{(J)}$ of N:
            1. If $Q^{(J)}$ is a placeholder node:
                1. Construct the subtree that represents the definition of production P of $Q^{(J)}$. The root of the new subtree is a rule node that represents the principal rule of P; each production referenced by the definition of P will be represented in the new subtree by a placeholder node.
                2. Set $Q^{(J)}$ to the root of the new subtree.
            2. $Q^{(J)}$ is a nonterminal rule node: TRAVERSE_NONTERMINAL

TRAVERSE_NONTERMINAL:

1. Pass (C, L, X) to $Q^{(J)}$.
2. Initialize the value $V_Q^{(J)}$ to *LACK*.
3. For each repetition node $H^{(J)}$ of $Q^{(J)}$ until $V_Q^{(J)}$ is *MORE*:
    1. Pass (C, L, X) to $H^{(J)}$.
    2. Initialize the value $V_H^{(J)}$ to *LACK*.
    3. For each choice node $N^{(J)}$ of $H^{(J)}$ until $V_H^{(J)}$ is *MORE*.
        1. Pass (C, L, X) to $N^{(J)}$.
        2. Initialize the value $V_N^{(J)}$ to *LACK*.
        3. Increment J.
        4. For each sequence node $Q^{(J)}$ of $N^{(J\text{-}1)}$ until $V_N^{(J\text{-}1)}$ is *MORE*:
            1. If $Q^{(J)}$ is a placeholder node:
                1. Construct the subtree that represents the definition of production P of $Q^{(J)}$. The root of the new subtree is a rule node that represents the principal rule of P; each production referenced by the definition of P will be represented in the new subtree by a placeholder node.
                2. Set $Q^{(J)}$ to the root of the new subtree.
            2. If $Q^{(J)}$ is a nonterminal principal rule node:
                1. If there exists queue Y in X that corresponds to production P of $Q^{(J)}$:
                    1. Using production P of $Q^{(J)}$, obtain from X queue Y that corresponds to P.
                    2. Add ($Q^{(J)}$) to Y.
                    3. Return $V_N^{(J)}$.
                2. Create (empty) queue Y.
                3. Add (P, Y) to X.
            3. If $Q^{(J)}$ is a nonterminal rule node:
                1. Recur to TRAVERSE_NONTERMINAL.
                2. If the return value $V_Q^{(J)}$ of $Q^{(J)}$ is *FULL*:
                    1. Using production P of $Q^{(J)}$, obtain from X queue Y that corresponds to P.
                    2. For each sequence node E dequed from Y until $V_Q^{(J)}$ is *MORE*:
                        1. Recur to TRAVERSE_NONTERMINAL, using E as $Q^{(J)}$.
                        2. Set $V_Q^{(J)}$ to the return value V_E of E.
                3. If the return value $V_Q^{(J)}$ of $Q^{(J)}$ is *FULL* and $V_N^{(J\text{-}1)}$ is *LACK*, set $V_N^{(J\text{-}1)}$ to $V_Q^{(J)}$. If the return value $V_Q^{(J)}$ of $Q^{(J)}$ is *MORE* and $V_N^{(J\text{-}1)}$ is not *MORE*, set $V_N^{(J\text{-}1)}$ to $V_Q^{(J)}$.
            4. If $Q^{(J)}$ is a terminal rule node:
                1. Pass (C, L) to $Q^{(J)}$.
                2. Initialize the value $V_P^{(J)}$ to *LACK*.
                3. For each repetition node $H^{(J)}$ of $Q^{(J)}$ until $V_Q^{(J)}$ is *MORE*: TRAVERSE_REPETITION

TRAVERSE_REPETITION:

1. Pass (C, L) to $H^{(J)}$.
2. Initialize the value $V_H^{(J)}$ to *LACK*.
3. For each choice node $N^{(J)}$ of $H^{(J)}$ until $V_H^{(J)}$ is *MORE*:
    1. Pass (C, L) to $N^{(J)}$.
    2. Initialize the value $V_N^{(J)}$ to *LACK*.
    3. Increment J.
    4. For each sequence node $Q^{(J)}$ of $N^{(J-1)}$ until $V_N^{(J)}$ is *MORE*:
        1. Pass (C, L) to $Q^{(J)}$.
        2. Initialize the value $V_Q^{(J)}$ to *LACK*.
        3. If $Q^{(J)}$ is a placeholder node:
            1. Construct the subtree that represents the definition of production P of $Q^{(J)}$. The root of the new subtree is a rule node that represents the principal rule of P; each production referenced by the definition of P will be represented in the new subtree by a placeholder node.
            2. Set $Q^{(J)}$ to the root of the new subtree.
        4. If C is a member of $K^{(J)}$:
            1. If $Q^{(J)}$ is a terminal principal rule node:
                1. If $Q^{(J)}$ is associated with a literal value $L^{(J)}$:
                    1. Set $V_Q^{(J)}$ to *FULL*.
                2. If the production P of $Q^{(J)}$ is associated with a concept $C^{(J)}$:
                    1. If $C^{(J)}$ mathces C:
                        1. If L is null:
                            1. If $D^{(J)}$ is not null:
                                1. Set $L^{(J)}$ to $D^{(J)}$.
                                2. Set $V_Q^{(J)}$ to *MORE*.
                        2. If L is not null:
                            1. Set $L^{(J)}$ to L.
                            2. Set $V_Q^{(J)}$ to *MORE*.
            2. If $V_Q^{(J)}$ is *LACK* and $Q^{(J)}$ is a terminal rule node:
                1. For each repetition node $H^{(J)}$ of $Q^{(J)}$ until $V_Q^{(J)}$ is *MORE*:
                    1. Recur to 3.3.4.2.3.3.4.4.3.1.
                    2. Set $V_Q^{(J)}$ to the return value $V_H^{(J)}$ of $H^{(J)}$.
                2. If $V_Q^{(J)}$ is *FULL* and $B^{(J)}$ is less than $U^{(J)}$:
                    1. Construct the subtree that represents the definition of the rule of $Q^{(J)}$. The root of the new subtree is a repetition node that represents the rule of $Q^{(J)}$; each production referenced by the definition of the rule of $Q^{(J)}$ will be represented in the new subtree by a placeholder node.
                    2. Add the root of the new subtree as a repetition node $H^{(J)}$ of $Q^{(J)}$.
                    3. Recur to 3.3.4.2.3.3.4.4.3.1.
                    4. Set $V_Q^{(J)}$ to the return value $V_H^{(J)}$ of $H^{(J)}$.
        5. If the return value $V_Q^{(J)}$ of $Q^{(J)}$ is *FULL* and $V_N^{(J-1)}$ is *LACK*, set $V_N^{(J-1)}$ to $V_Q^{(J)}$. If the return value $V_Q^{(J)}$ of $Q^{(J)}$ is *MORE* and $V_N^{(J-1)}$ is not *MORE*, set $V_N^{(J-1)}$ to $V_Q^{(J)}$.
    5. If the return value $V_N^{(J)}$ of $N^{(J)}$ is *FULL* and $V_H^{(J-1)}$ is *LACK*, set $V_H^{(J-1)}$ to $V_N^{(J)}$. If the return value $V_N^{(J)}$ of $N^{(J)}$ is *MORE* and $V_H^{(J-1)}$ is not *MORE*, set $V_H^{(J-1)}$ to $V_N^{(J)}$.
    6. Decrement J.

C.2. Composition

Once an ACT has been constructed, the extraction of a message is relatively straightforward. A pretraversal of the each message subtree is performed, and the first subtree that returns a non-null message string will be the message composed (and thus sent).

An extra stack for whitespace separators must be maintained; the size of this stack corresponds to the depth of the tree during the traversal. The top element of the stack will be appended to each token that is added to a composition. Though such a separator is not always necessary (as discussed in Section 5.3.9), always appending the relevant separator after every token circumvents a great deal of analysis over the set of tokens, and does not significantly hinder a parse of the composition.

The other special composition consideration relates to the discovery of an unresolved placeholder node in the ACT. When such a node is discovered, it must be (non-recursively) expanded according to the above algorithm in order to determine if any default literal values included in the expansion allow the subtree replacement of the placeholder to participate in the composition.

Appendix D — Protocol Specifications

This appendix details the protocol specifications that were used for the experiments described in Chapter 7. Note that the presence of these specifications in this document are considered to be merely a convenient reference, and should not be consulted as a thorough study of the protocols in question. Further, since a concept specifications is merely a list of concept names, they are considered implied and consequently not shown.

D.1. HTTP

Two versions of a minimum HTTP protocol were specified, one representing HTTP 1.0 and one representing HTTP 1.1. Each is specified as a collection of files.

D.1.1. HTTP 1.0

The file `top`:

```
[interactions]
  GET
  POST
[end]
```

The file `itr.GET`:

```
[messages]
  REQUEST_GET
  RESPONSE_200
  RESPONSE_404
[roles]
  org.w3c.http 1.0 base
[states]
  org.w3c.http 1.0 base
[initializations]
  CLIENT START
  SERVER START
[communications]
  REQUEST_GET
  : {START} CLIENT REQUESTED
 -> {START} SERVER REQUESTED
  RESPONSE_200
  : {REQUESTED} SERVER RESPONDED
 -> {REQUESTED} CLIENT RESPONDED
  RESPONSE_200
  : {REQUESTED} SERVER RESPONDED
 -> {REQUESTED} CLIENT RESPONDED
[end]
```

The file `itr.POST`:

```
[messages]
  REQUEST_POST
  RESPONSE_200
  RESPONSE_404
[roles]
  org.w3c.http 1.0 base
[states]
  org.w3c.http 1.0 base
[initializations]
  CLIENT START
  SERVER START
[communications]
  REQUEST_POST
  : {START} CLIENT REQUESTED
 -> {START} SERVER REQUESTED
  RESPONSE_200
  : {REQUESTED} SERVER RESPONDED
 -> {REQUESTED} CLIENT RESPONDED
  RESPONSE_404
  : {REQUESTED} SERVER RESPONDED
 -> {REQUESTED} CLIENT RESPONDED
[end]
```

The file `msg.REQUEST_GET`:

```
[tokens]
  org.w3c.http 1.0 base
  org.w3c.uri 1.0 base
[structures]
  org.w3c.http 1.0 header
[concepts]
  org.w3c.http 1.0 base
[productions]
  LINE (
    <METHOD_GET_ID> <SPACE> <URI> <SPACE> <PROTOCOL>
  )
  REQUEST_GET (LINE <CRLF> <CRLF>) = REQUEST_DOWNLOAD
[end]
```

The file `msg.REQUEST_POST`:

```
[tokens]
  org.w3c.http 1.0 base
  org.w3c.uri 1.0 base
[structures]
  org.w3c.http 1.0 header
[concepts]
  org.w3c.http 1.0 base
[productions]
  LINE (
    <METHOD_POST_ID> <SPACE> <URI> <SPACE> <PROTOCOL>
  )
  REQUEST_POST (
    LINE <CRLF> <BODY> <CRLF>
  ) = REQUEST_ACTION
[end]
```

The file `msg.RESPONSE_200`:

```
[tokens]
  org.w3c.http 1.0 base
  org.w3c.http 1.0 body
[structures]
  org.w3c.http 1.0 header
[concepts]
  org.w3c.http 1.0 base
[productions]
  LINE (<PROTOCOL> <SPACE> <STATUS_200_ID> <SPACE> <OK>)
  RESPONSE_200 (
    LINE <CRLF> HEADER_LIST <CRLF> <BODY>
  ) = RESPONSE_SUCCESS
[end]
```

The file `msg.RESPONSE_404`:

```
[tokens]
  org.w3c.http 1.0 base
  org.w3c.http 1.0 body
[structures]
  org.w3c.http 1.0 header
[concepts]
  org.w3c.http 1.0 base
[productions]
  LINE (
    <PROTOCOL> <SPACE>
    <STATUS_404_ID> <SPACE>
    <NOT_FOUND>
  )
  RESPONSE_404 (
    LINE <CRLF> HEADER_LIST <CRLF> <BODY>
  ) = RESPONSE_SUCCESS
[end]
```

The file `stc.header`:

```
[tokens]
  org.w3c.http 1.0 base
  org.w3c.http 1.0 header
[productions]
  CONTENT_LENGTH (
    <CONTENT_LENGTH_ID> <SEP> <CONTENT_LENGTH_VALUE>
  )
  HEADER (CONTENT_LENGTH)
  HEADER_LIST (HEADER <CRLF>)*
[end]
```

The file `tkn.header`:

```
[concepts]
  org.w3c.http 1.0 header
[productions]
  SEP (": ")
  CONTENT_LENGTH_ID ("Content-Length")
  CONTENT_LENGTH_VALUE $optional (
    "0" | ["1"-"9"](["0"-"9"])*
  ) = CONTENT_LENGTH
[end]
```

The file `tkn.body`:

```
[tokens]
  org.w3c.http 1.0 header
[concepts]
  org.w3c.http 1.0 body
[productions]
  CONTENT ([" "-"~"]) = ATOM
  BODY ((<CONTENT>) {<CONTENT_LENGTH_VALUE>}) = BODY
[end]
```

The file `tkn.base`:

```
[concepts]
  org.w3c.http 1.0 base
[productions]
  SPACE (" ")
  CRLF (("\r")? "\n")
  METHOD_GET_ID $optional $constant (
    "GET"
  ) = METHOD_GET_NAME
  METHOD_POST_ID $optional $constant (
    "POST"
  ) = METHOD_POST_NAME
  STATUS_200_ID $optional $constant (
    "200"
  ) = STATUS_200_NAME
  STATUS_404_ID $optional $constant (
    "404"
  ) = STATUS_404_NAME
  PROTOCOL_ID $constant ("HTTP")
  PROTOCOL_VERSION_MAJOR_ID $constant (
    "1"
  ) = VERSION_MAJOR
  PROTOCOL_VERSION_MINOR_ID $constant (
    "0"
  ) = VERSION_MINOR
  PROTOCOL_VERSION $constant (
    <PROTOCOL_VERSION_MAJOR_ID> "."
    <PROTOCOL_VERSION_MINOR_ID>
  )
  PROTOCOL $constant (
    <PROTOCOL_ID> "/" <PROTOCOL_VERSION>
  ) = PROTOCOL
  OK $constant ("OK")
  NOT_FOUND $constant ("Not Found")
[end]
```

D.1.2. HTTP 1.1

The specification of the selected elements of HTTP 1.1 differs only slightly from that of HTTP 1.0. Therefore, enumerating the files of the specification for HTTP 1.1 would be redundant.

D.2. SIENA

Two versions of the complete SIENA protocol were specified, one representing SIENA 1.4 and one representing SIENA 1.5. Each is specified as a collection of files.

D.2.1. SIENA 1.4

Because the interactions and messages of SIENA are nearly identical, only the specification of a representative of each category is shown.

The file `top`:

```
[interactions]
  DELIVERY
  PUBLICATION
  SUBSCRIPTION
  UNSUBCRIPTION
  BYE
  SUSPEND
  RESUME
  MAP
  CONFIGURE
  SHUTDOWN
  WHO
[end]
```

The file `itr.PUBLICATION`:

```
[messages]
  PUBLICATION
[roles]
  siena 1.4.3 base
[states]
  siena 1.4.3 base
[initializations]
  CLIENT START
  SERVER START
[communications]
  PUBLICATION : {START} CLIENT END -> {START} SERVER END
[end]
```

The file `msg.PUBLICATION`:

```
[tokens]
  siena 1.4.3 header = HEADER
[structures]
  siena 1.4.3 header = HEADER
  siena 1.4.3 event
[concepts]
  siena 1.4.3 base
[productions]
  HEADER (
    <NAME@HEADER> <OPEN@HEADER>
    (VERSION@HEADER)? METHOD_PUB@HEADER TO@HEADER
    <CLOSE@HEADER>
  )
  PUBLICATION (HEADER (EVENT | EVENTS)) = PUBLICATION
[end]
```

The file `stc.header`:

```
[tokens]
  siena 1.4.3 header
[productions]
  METHOD_PUB (<METHOD_NAME> <SEP> <METHOD_VALUE_PUB>)
  METHOD_SUB (<METHOD_NAME> <SEP> <METHOD_VALUE_SUB>)
  METHOD_UNS (<METHOD_NAME> <SEP> <METHOD_VALUE_UNS>)
  METHOD_BYE (<METHOD_NAME> <SEP> <METHOD_VALUE_BYE>)
  METHOD_SUS (<METHOD_NAME> <SEP> <METHOD_VALUE_SUS>)
  METHOD_RES (<METHOD_NAME> <SEP> <METHOD_VALUE_RES>)
  METHOD_MAP (<METHOD_NAME> <SEP> <METHOD_VALUE_MAP>)
  METHOD_CNF (<METHOD_NAME> <SEP> <METHOD_VALUE_CNF>)
  METHOD_OFF (<METHOD_NAME> <SEP> <METHOD_VALUE_OFF>)
  METHOD_WHO (<METHOD_NAME> <SEP> <METHOD_VALUE_WHO>)
  METHOD_INF (<METHOD_NAME> <SEP> <METHOD_VALUE_INF>)
  TO (<TO_NAME> <SEP> <TO_VALUE>)
  ID (<ID_NAME> <SEP> <ID_VALUE>)
  HANDLER (<HANDLER_NAME> <SEP> <HANDLER_VALUE>)
  VERSION (<VERSION_NAME> <SEP> <VERSION_VALUE>)
[end]
```

The file `stc.event`:

```
[tokens]
  siena 1.4.3 event
[productions]
  EVENT (<EVENT_NAME> <OPEN> (<ATTRIBUTE>)+ <CLOSE>)
  EVENTS (<EVENTS_NAME> <OPEN> (EVENT)+ <CLOSE>)
[end]
```

The file `tkn.header`:

```
[tokens]
  siena 1.4.3 base = BASE
[concepts]
  siena 1.4.3 header
[productions]
  NAME ("senp")
  OPEN (<OPEN@BASE>)
  CLOSE (<CLOSE@BASE>)
  SEP (<SEP@BASE>)
  METHOD_NAME ("method")
  METHOD_VALUE_PUB $optional $constant (
    "PUB"
  ) = METHOD_PUB
  METHOD_VALUE_SUB $optional $constant (
    "SUB"
  ) = METHOD_SUB
  METHOD_VALUE_UNS $optional $constant (
    "UNS"
  ) = METHOD_UNS
  METHOD_VALUE_BYE $optional $constant (
    "BYE"
  ) = METHOD_BYE
  METHOD_VALUE_SUS $optional $constant (
    "SUS"
  ) = METHOD_SUS
  METHOD_VALUE_RES $optional $constant (
    "RES"
  ) = METHOD_RES
  METHOD_VALUE_MAP $optional $constant (
    "MAP"
  ) = METHOD_MAP
  METHOD_VALUE_CNF $optional $constant (
    "CNF"
  ) = METHOD_CNF
  METHOD_VALUE_OFF $optional $constant (
    "OFF"
  ) = METHOD_OFF
  METHOD_VALUE_WHO $optional $constant (
    "WHO"
  ) = METHOD_WHO
  METHOD_VALUE_INF $optional $constant (
    "INF"
  ) = METHOD_INF
  TO_NAME ("to")
  TO_VALUE none (<ID>) = TO
  ID_NAME ("id")
  ID_VALUE none (<ID>) = ID
  HANDLER_NAME ("handler")
  HANDLER_VALUE none (<URL>) = HANDLER
  VERSION_NAME ("version")
  VERSION_VALUE $constant ("1.5.0") = VERSION
[end]
```

The file `tkn.event`:

```
[tokens]
  siena 1.4.3 base = BASE
[concepts]
  siena 1.4.3 event
[productions]
  EVENT_NAME ("event")
  EVENTS_NAME ("events")
  OPEN (<OPEN@BASE>)
  CLOSE (<CLOSE@BASE>)
  SEP (<SEP@BASE>)
  ATTRIBUTE $none (<NAME> <SEP> <VALUE>) = ATTRIBUTE
  VALUE $none (
    <NULL_VALUE>
  | <BOOLEAN_VALUE> | <INTEGER_VALUE>
  | <DECIMAL_VALUE> | <STRING_VALUE>
  )
  NULL_VALUE (<NULL@BASE>)
  BOOLEAN_VALUE (
    <BOOLEAN_VALUE_TRUE> | <BOOLEAN_VALUE_FALSE>
  ) = BOOLEAN_VALUE
  BOOLEAN_VALUE_TRUE none (
    <TRUE@BASE>
  ) = BOOLEAN_VALUE_TRUE
  BOOLEAN_VALUE_TRUE none (
    <TRUE@BASE>
  ) = BOOLEAN_VALUE_FALSE
  INTEGER_VALUE $none (<INTEGER@BASE>) = INTEGER_VALUE
  DECIMAL_VALUE $none (<DECIMAL@BASE>) = DECIMAL_VALUE
  STRING_VALUE $none (
    <QUOTE@BASE> <STRING@BASE> <QUOTE@BASE>
  ) = STRING_VALUE
[end]
```

The file `tkn.base`:

```
[productions]
  OPEN ("{")
  CLOSE ("}")
  SEP ("=")
  NAME (
    ["A"-"Z", "a"-"z", "_"]
    ["0"-"9", "A"-"Z", "a"-"z", "_", ".", "/", "$"]*
  )
  NULL ("null")
  TRUE ("true")
  FALSE ("false")
  INTEGER ("0" | ["1"-"9"](["0"-"9"])*)
  DECIMAL (
    ("0" | ["1"-"9"](["0"-"9"])*) "." (["0"-"9"])+
  )
  QUOTE ("\"")
  STRING ([" "-"!", "#"-"[", "]"-"~"] | "\\\"" | "\\\\")
[end]
```

D.2.2. SIENA 1.5

As with HTTP, the specification of the selected elements of SIENA 1.4 differs only slightly from that of SIENA 1.5 (the only substantial difference being the deprecation of the WHO interaction). Therefore, enumerating the files of the specification for SIENA 1.5 would be redundant.