Composing Message Translators and Inferring their Data Types using Tree Automata

Emil - Mircea Andriescu, Thierry Martinez, Valérie Issarny

Composing Message Translators and Inferring their Data Types using Tree Automata

Emil Andriescu, Thierry Martinez, Valérie Issarny

Inria Paris-Rocquencourt,
Domaine de Voluceau, Rocquencourt, Le Chesnay 78153, France

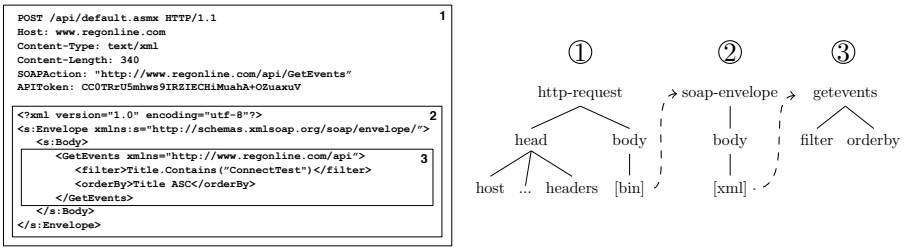
Abstract. Modern distributed systems and Systems of Systems (SoS) are built as a composition of existing components and services. As a result, systems communicate (either internally, locally or over networks) using protocol stacks of ever-increasing complexity whose messages need to be translated (i.e., interpreted, generated, analyzed and transformed) by third-party systems such as services dedicated to security or interoperability. We observe that current approaches in software engineering are unable to provide an efficient solution towards reusing message translators associated with the message formats composed in protocol stacks. Instead, developers must write ad hoc "glue-code" whenever composing two or more message translators. In addition, the data structures of the output must be integrated/harmonized with the target system.

In this paper we propose a solution to the above that enables the composition of message translators according to a high-level user-provided query. While the composition scheme we propose is simple, the inference of the resulting data structures is a problem that has not been solved up to now. This leads us to contribute with a novel data type inference mechanism, which generates a data-schema using tree automata, based on the aforementioned user query.

1 Introduction

Protocols used by modern systems are becoming increasingly complex, while standards bodies are unable to keep pace with the current speed of tech development [16]. At the same time, there is an emerging need to analyze interactions and facilitate interoperability of new as well as legacy systems in the absence of the protocol stack implementations. This need originates from the way systems are designed. Indeed, most modern systems are productive and cost-effective only if they can interoperate with other systems, sharing with them data and functionalities [9]. Additionally, analyzing system interactions at run-time is vital in assuring security in enterprise environments (e.g., protocols in use should be made compatible with corporate firewalls), but can also help in achieving interoperability by automatically discovering/learning functional aspects of the system, such as: application behavior [11], data semantics [10], etc.

Data formats, and in particular message formats, have long represented a barrier to interoperability [3]. This is because software parts often make different assumptions about how data is represented [21]. A crosscutting challenge is further represented by *protocol binding*, i.e., the way protocols are combined to form



(a) Network message (b) Corresponding ASTs

Fig. 1: A composed message sample and its corresp. ASTs (leaf nodes omitted)

a protocol stack, which causes systems to exchange messages combining multiple syntaxes (i.e., composite message formats). The resulting message formats can include a mix of text encodings, binary encodings and data serialization formats. As an example, consider the message depicted in Figure 1a. The message combines two distinct data representations: ① a text-based message part that corresponds to the HTTP protocol, and parts ② & ③ that use XML serialization. In the case where individual translators for each independent message format are available (e.g., for parts ①, ② and ③), "glue code" has to be provided in order to compose them. However, to the best of our knowledge, existing methods for the composition of parsers are highly specific to a given parsing method or algorithm (e.g., grammar composition [19], parser-combinators [13], parse-table composition [20]) and cannot be easily generalized. As a result, existing methods do not allow for the systematic composition of message translators out of third-party translators, thereby requiring developers to implement hardcoded adapters in order to process messages.

Additionally, the problem of composing message translators is related to the composition of the associated data structures (or inference of composite data types). The composition of data structures relative to the composition of message translators ensures that composite translators can be seamlessly (or even automatically) integrated with existing systems. For instance, when parsing the message in Figure 1a using a composite translator we would normally obtain three separate data structures, as shown in Figure 1b in the form of Abstract Syntax Trees (AST). Knowledge of the precise constraints on the structure of ASTs is essential if composite translators are to be integrated into other systems. For instance, in Enterprise Service Busses (ESB) [6], service adaptation requires the transformation of messages into a uniform representation (most commonly XML). However, to the best of our knowledge, in all existing ESB implementations, XML data-schemas must be hand-coded by the developers of adaptors. AST composition can be arbitrary, but it should not result in the loss of information. In other words, the AST transformation applied for the composition must be injective, thus invertible. In the context of this paper, we are particularly interested in the *substitution* class of AST transformations, that is, the substitution of a leaf node by a sub-tree. This allows us to represent encapsulated message formats in a hierarchical manner. In Figure 1b, we exemplify such a case of substitution using dotted arrows, as follows: (i) the node labeled

[*bin*] in AST ① is substituted by AST ②, and (ii) the node labeled [*xml*] in AST ② is substituted by AST ③. While other compositions are possible (e.g., AST ② could alternatively be appended to the root of AST ①), this particular one closely resembles the way most protocols arrange encapsulated data, therefore being the most intuitive. The tree transformation mentioned above can be easily expressed by adapting already existing mechanisms for XML, such as the XSLT transformation language in combination with the XPath query language. In general, defining the composition of ASTs is rather straightforward. However, inferring the data structure resulting from an AST composition is more complex Indeed, it is already known that for an arbitrary tree transformation, the problem of type inference may not have a solution [12]. While the problem of type inference is quite common in the domains of functional programming languages [2,5] and XML technologies [12,17,18], we are not aware of any solution capable of type inference for the *substitution* class of tree compositions although this kind of transformation is very common in practice (most notably in the XML transformation language XSLT).

Following the above, this paper makes two major contributions to the issue of systematic message translation for modern distributed systems:

1. Starting from the premise that "off-the-shelf" message translators for individual protocols are readily available in at least an executable form, we propose a solution for the automated composition of message translators. The solution simply requires the specification of a composition rule that is expressed using a subset of the navigational core of the W3C XML query language XPath [8].
2. We provide a formal mechanism, using tree automata, which based on the aforementioned composition rule, generates an associated AST *data-schema* for the translator composition. This contribution enables the inference of correct data-schemas, relieving developers from the time-consuming task of defining them. On a more general note, the provided method solves the type inference problem for the *substitution* class of tree compositions in linear time on the size of the output. The provided inference algorithm can thus be adapted to a number of applications beyond the scope of this paper, such as XML Schema inference for XSLT transformations.

There is already a number of systems that can benefit from translator composition such as: Packet Analyzers, Internet Traffic Monitoring, Vulnerability Discovery, Application Integration and Enterprise Service Bus, etc. As a result, our approach can have an immediate impact knowing that current implementations rely on tightly coupled and usually hardcoded message translators, to the detriment of software reuse.

The paper is organized as follows. The next section provides the background of our research focusing on challenges related to message parsing and translator composition, as well as to the problem of AST data-schema inference for a given composition rule. Then, Section 3 details our approach to the systematic composition of message translators, while Section 4 introduces a method to automatically generate AST data-schemas (formalized as Hedge Automata) associated to a given composition of translators. Section 5 assesses our current

prototype implementation and its benefits. Finally, Section 6 summarizes our contributions over related work and introduces perspectives for future work.

2 Background

Ideally, message translators may be developed by separate parties, using various technologies, while developers should be able to compose them using an easy to use mechanism. A common challenge in realizing such a mechanism is related to parsing composite message formats. Specifically, one must identify the mechanisms of message encapsulation, that is, the rules upon which one message syntax is combined with another message syntax. However, *"parsers are so monolithic and tightly constructed that, in the general case, it is impossible to extend them without regenerating them from a source grammar"* [20]. Even if the source grammars are made available, composition is still an issue, taking into account that combining two unambiguous grammars may result in an ambiguous grammar, and that the ambiguity detection problem for context-free grammars is undecidable in the general case [1].

2.1 Encapsulated Message Formats

The following investigates in more detail the various cases, from the most complex (also the most general) to the most trivial, of syntax composition to highlight the specific case applying to message composition.

Mutually-recursive syntax composition refers to the case where the syntax of two distinct message formats can mutually be included inside one another. A technique commonly used to support this case of composition is *recursive descent parsing* (in particular implemented by parser combinators [22,13]), where a composite parser is defined from a set of mutually recursive procedures. This class of syntax composition has been extensively studied in the domain of extensible programming languages [19,20], where parser composition allows extending the syntax of a "host" programming language, such as Java, with an "extension", such as SQL. Intuitively, the syntax is mutually-recursive because SQL queries can appear within Java expressions, and, at the same time, Java expressions can appear within SQL queries, allowing an unbounded chain of compositions. The same cannot be said about messages exchanged by protocol stacks where mutually-recursive compositions are unlikely given the fixed number of layers.

Syntax composition represents a restriction of the case above where the inclusion is not mutual. This class includes both *syntax extensions*, which allow expanding the initial language with new message types, as well as *syntax restrictions*, which introduce intentional limitations on the expressiveness of a language, in the same sense explained by Cardelli *et al.* in [4]. The way two syntaxes can be composed to actually reduce the expressiveness of a language is by giving new constraints for existing terminal symbols. For instance, the message in Figure 1a can be successfully parsed by a HTTP parser or by a composed HTTP/SOAP parser. The difference is that the latter will only allow messages that contain SOAP messages in the HTTP *body* section.

Stratified syntax inclusion is a special case of syntax composition, which may only restrict the expressiveness of the base language. Commonly, a middleware protocol parser is syntactically "unaware" of encapsulated messages, which are treated as a collection of binary data or arbitrary character strings. Informally, we can say that message parsers should be "forward-compatible" with respect to possible message compositions. Because of this containment property, we can state that whenever two message translators are composed to handle an encapsulated message format, they specialize (or restrict) the set of messages initially accepted. Figure 1a illustrates a concrete example of *stratified syntax inclusion*, where a *SOAP-envelope* message ② is included in the *body* section of an *HTTP request* message ①. The second encapsulation between layers ② and ③ is slightly different because both messages share the same lexical syntax (i.e., XML), but have different data syntax (formulated in terms of XML).

Trivial inclusion refers to the case where the syntax of composed message formats is delimited by other, more trivial, means. For instance, some protocols compose messages by simply arranging the content in a sequential manner (e.g., one parser analyzes a part of the input, and returns the remaining part in its result). This kind of composition can be easily modeled as a particular case of *stratified syntax inclusion*.

Considering the above cases, we narrow our problem to the case of *stratified syntax inclusion*. While this type of syntax composition is generic enough to capture encapsulation mechanisms implemented by most protocol stacks, it also allows avoiding parsing ambiguity without difficulty. Thanks to the characteristics of stratified syntax inclusion, compositions may be implemented without knowing the internal aspects of translators, as we further present in Section 3.

2.2 Data Type Inference

As far as we know, the problem of inferring the output schema (or the data type) of an arbitrary tree transformation has not yet been solved, while it is known that, in general, a transformation might not be recognizable by a schema [12].

In [12], Milo *et al.* propose an approach capable of type inference for queries over semistructured data, and in particular XML. However, the query mechanism for node selection that is proposed is only capable of vertical navigation, meaning that the language does not allow conditions on the ordering of nodes (horizontal navigation), which is particularly required for selecting ordered nodes of an AST. In [17], Papakonstantinou *et al.* propose an improved approach that can infer Data Type Definitions (DTDs) for views of XML documents. In their work, *views* are in fact subtrees that are selected using a query language capable of both vertical and horizontal selection conditions. The solution can be easily generalized to select *views* from multiple trees/sources. However, it is not capable of merging the results from different XML languages, as it is required in the case of translator composition.

ℂDuce [2] is a language for expressing well-typed XML transformations. Specifically, ℂDuce can automatically infer non-recursive data types corresponding to a provided XML transformation. ℂDuce does not propose a high-level tree composition mechanism, but rather provides a language where XML queries and

transformations can be implemented using a low-level form of navigational patterns that are non-compliant with the XPath standard. In [5], an extension was provided, which essentially allows implementing XPath-like navigation expressions by pattern matching and to precisely type them. While this improves the query mechanism, it does not solve the limitations of the transformation mechanism which, in our understanding, is limited to disjoint trees concatenated in the result. We mention that the problem of type inference is related, but different to the problem of static type checking for XSLT transformations [23,18], which intends to verify that a program always converts valid source documents into also valid output documents for the case were both input and output schemas are known.

Considering the above, none of the approaches solve the AST type inference problem for the *substitution* class of transformations, which applies whenever two or more message translators are combined. In the next section, we introduce our approach for translator composition and detail the *substitution* transformation that is applied.

3 Message Translator Composition

Translator composition includes two mutually-dependent problems, the composition of parsers and the composition of un-parsers (also referred to as message composers). Because of space restrictions, we only discuss the first one in detail, while the latter can be easily deduced from our presentation.

Definition 1 (Message translator). *A message translator comprises a parsing function $P : M \to T(\Sigma)$ that takes as input a bit-string and outputs a tree, and the inverse $C = P^{-1}$ where:*

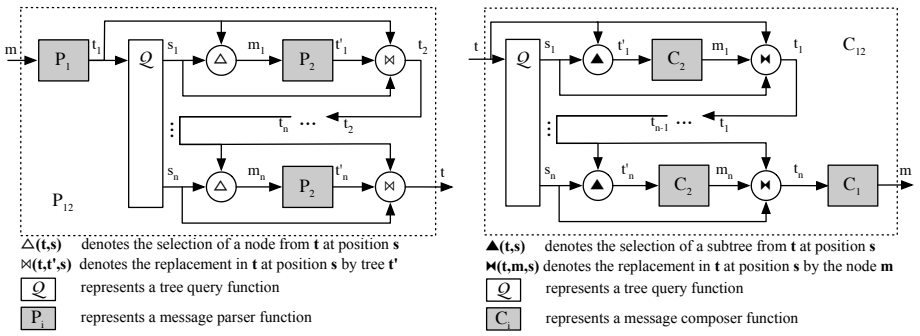
- $M \subseteq \{0,1\}^*$
- $T(\Sigma)$ *is a set of finite ordered, unranked, directed, and rooted trees*[1] *labelled over the finite alphabet $\Sigma = \Sigma_0 \cup \{\beta, 0, 1\}$, where Σ_0 is a set of labels, not including the set of* binary labels $\{0,1\}$*, neither the* binary-subtree label β[2].

In Figure 2, we detail our method of composing message translators in the form of two block diagrams, corresponding to the composition of, respectively, parser functions and the inverse composer functions.

Parser composition requires three inputs, (i) a parser P_1, (ii) a query $\mathcal{Q}$ that identifies parts of the output of P_1 which contains the encapsulated data, encoded in an auxiliary message format, and a parser P_2 that corresponds to the auxiliary message format. Informally, the method works as follows. First, the stratified

[1] A finite ordered tree t over a set of labels Σ is a mapping from a finite prefix-closed set of positions $\mathcal{P}os(t) \subseteq N^*$ into Σ as explained in [7].

[2] Note that since elements of T represent ASTs, arbitrary data can be included only as leaf nodes, in the form of an ordered sequence of binary labels, by convention, under a β-labeled node. Such a structure (e.g., $\beta \to 1011...$) is equivalent to a bit-string $b \in \{0,1\}^*$. We use this convention to avoid having an infinite label alphabet, such as $\Sigma_0 \cup \{0,1\}^*$, that would be outside the scope of regular tree automata theory.



(a) Parser (b) Composer

Fig. 2: Message translator composition mechanism

input message is parsed using the parser P_1, which corresponds to the first stratum of the message. The query $\mathcal{Q}$ is then used to select positions in the resulting AST which correspond to encapsulated messages (of a second format). Then, the AST of the composite message is obtained by substituting in the initial tree every position that belongs to an answer to $\mathcal{Q}$ by trees resulting from the parsing of the encapsulated messages using P_2.

To better exemplify this mechanism, we consider the stratified message presented in Figure 1 (the leaf nodes containing message data values are omitted). The first stratum (or layer) ① of the message consists of a HTTP request message. By passing this message through a HTTP translator, we obtain an AST representation similar to Tree ①. Knowing that the sub-tree [*bin*] attached to the *body* node contains (encapsulates) SOAP message syntax we may pass this data to a SOAP translator to be interpreted. To support this kind of composition for all trees of the form of Tree ①, which may be an infinite set, we must generalize the composition mechanism. To do so, we can define the node-selection requirements as a *unary* (or *node-selecting*) *tree query* [18].

In the context of this paper, we consider a tree-query to be a subset of the navigational core of the W3C XML query language XPath [8], which we represent formally in Section 4 as *tree query automata*. Using the XPath syntax, we can write $T_{HTTP}[/request/body/] \rightarrow T_{SOAP}$, meaning that the translator T_{HTTP} is composed with translator T_{SOAP} such that, for a given composite message, all nodes selected by the query $/request/body/$ are substituted with an AST corresponding to T_{SOAP}. While this example is trivial, more complex queries are supported. For instance, defining the composition between a HTTP translator and a MIME-type translator can be specified as $T_{HTTP}[/request/head/header[key =' Content - Type']/value] \rightarrow T_{MIME}$, making use of an XPath predicate that enables the selection of a node that contains a specific value.

Formally, we introduce the following definition for parser composition:

Definition 2 (Parser composition). *Given two message parsers $P_1 : M_1 \rightarrow T_1(\Sigma_1)$, $P_2 : M_2 \rightarrow T_2(\Sigma_2)$ and a user-defined tree query $\mathcal{Q}$ for $t_1 \in T_1(\Sigma_1)$, we define the composed parser $P_{12} : M_{12} \rightarrow T_{12}(\Sigma_1 \cup \Sigma_2)$ as follows (see Figure 2a):*

— *For a stratified message* $m \in M_1$*, we apply the query* $\mathcal{Q}$ *on* t_1*, where* $t_1 = P_1(m)$*.*
— *The answer to* $\mathcal{Q}$ *for* t_1 *is* $S = \{s_1, ..., s_n\}$*, the set of selected positions in the tree* t_1*, with* $n \geq 0$*.*
— *For each* $s_i \in S$*, we compute* $t_{i+1} = \bowtie(t_i, P_2(\triangle\ (t_i, s_i)), s_i)$ *where:*
  — $\triangle(t, s)$ *denotes the selection of a bit-string from* t *at position* s*;*
  — $\bowtie(t, t', s)$ *denotes the replacement in* t *of a bit-string at position* s *by* t'*.*
— *The composed parser function* $P_{12} : M_{12} \rightarrow T_{12}(\Sigma_1 \cup \Sigma_2)$*, with* $M_{12} \subseteq M_1$*, is defined as* $P_{12}(m) = t_{n+1}$ *(*$t_{n+1} = t$ *in Figure 2a).*

Similarly, Figure 2b illustrates the inverse function (i.e., composer composition, whose definition is direct to infer from Definition 2). In the compositions of both functions, we consider that the elements of the query result $S = \{s_1, ..., s_n\}$ are *prefix-disjoint*, meaning that for any position s_i of the form $s_i = s_j s'$, then $i = j$. This property ensures that the *selection* $\triangle(t, s)$ and *replacement* $\bowtie(t, t', s)$ operations for a query result S on a *tree* t can be performed in an arbitrary order.

We observe that the aforementioned composition method is part of a wider class of *result-conversion* mechanisms. Most notably, the Scala (`http://www.scala-lang.org/`) programming language implements a result-conversion *parser combinator*. A result-conversion combinator, denoted $P\hat{}\hat{}f$, is defined as a higher-order function, which takes as input a parser function P and a user-defined function f that is applied on the result of P. The modified parser succeeds exactly when P succeeds. This method is particularly relevant to our case because it is purely defined on the output data type, and thus it does not require any knowledge about the input message syntax. However, in our case, the function f is not arbitrary, since it is represented by a user query.

By applying the composition mechanism defined above, we are able to compose message translators as black-box functions, which, in turn, allows the translation of composite messages (for the case of *stratified syntax inclusion*) to and from a uniform tree representation. However, the constraints on the structure of this tree representation (i.e., the data-schema) are unknown. In the next section, we provide a formal mechanism by which we are able to automatically generate a *data-schema* for the resulting ASTs.

4 Data-schema Composition

Data-schema languages share unranked tree automata as theoretical foundation [15]. In what follows, we use top-down non-deterministic finite hedge automata [14] (or, equivalently, hedge grammars) to model AST languages. Below, we first recall the basic definitions associated with tree automata.

4.1 Definitions

Definition 3 (NFHA). *A top-down non-deterministic finite hedge automaton (NFHA) is a tuple* $\mathcal{A} = (Q, \Sigma, Q_0, \Delta)$ *where* Q *is a finite set of states,* Σ *is an alphabet of symbols,* $Q_0 \subseteq Q$ *is a subset of accepting states, and* Δ *is a finite set of transition rules of the type* $q \rightarrow a(R)$ *where* $q \in Q$*,* $a \in \Sigma$*, and* $R \subseteq Q^*$ *is a regular language over* Q*.*

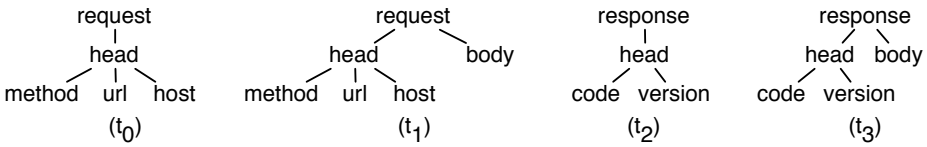


Fig. 3: Sample ASTs (leaf and β nodes omitted)

Additionally, we introduce the definition of $\mathcal{A}$-derivations, which we use as a helper mechanism to describe the process of tree evaluation by a hedge automaton.

Definition 4 ($\mathcal{A}$-derivations). *Given an automaton $\mathcal{A}$, $\mathcal{A}$-derivations are defined inductively as follows. A tree $r \in T(Q)$ with $r = q(r_1 \dots r_n)$ is a derivation from a state $q \in Q$ for a tree $t \in T(\Sigma)$ with $t = a(t_1 \dots t_n)$ if:*

— *There exists a transition rule $q \to a(R) \in \Delta$ such that $q_1 \dots q_n \in R$,*
— *For all $1 \leq i \leq n$, r_i is an $\mathcal{A}$-derivation from $q_i \in Q$ for t_i.*

A tree $t \in T(\Sigma)$ is accepted by an automaton $\mathcal{A}$ if there exists a derivation from a state $q_0 \in Q_0$ for t.

Example 1 (NFHA) Consider the automaton $\mathcal{A}^b = (Q^b, \Sigma^b, Q_0^b, \Delta^b)$ that recognizes the language $L(\mathcal{A}^b) = \{t_0, t_1, t_2, t_3\}$ containing the trees shown in Figure 3. The automaton is defined as follows:

— $Q^b = \{q_0, q_1, ..., q_{10}\}$, $Q_0^b = \{q_0, q_1\}$,
— $\Sigma^b = \{request, response, head, body, ...\} \cup \{\beta, 0, 1\}$,
— $\Delta^b = \{q_0 \to request(q_3\, q_2?), q_1 \to response(q_4\, q_2?), q_2 \to body(q_{10}),$
$q_3 \to head(q_5\, q_6\, q_7), q_4 \to head(q_8\, q_9), q_5 \to method(q_{10}), q_6 \to url(q_{10}),$
$q_7 \to host(q_{10}), q_8 \to code(q_{10}), q_9 \to version(q_{10}), q_{10} \to \beta(q_\beta)\}$,
q_β is a state that accepts any sequence of $\{0, 1\}$ leaves.

The $\mathcal{A}_b$-derivation for the tree $t_0 = request\langle head\langle method\, url\, host\rangle\rangle$ (shown in Figure 3) is the tree $r = q_0\langle q_3\langle q_5\langle q_{10}\langle q_\beta\rangle\rangle\, q_6\langle q_{10}\langle q_\beta\rangle\rangle\, q_7\langle q_{10}\langle q_\beta\rangle\rangle\rangle\rangle$, where $r \in T(Q^b)$, thus, we can say that t_0 is accepted by $\mathcal{A}^b$.

While data-schemas are formalized as NFHA, we introduce a second type of tree automata, which we use to formalize tree queries. Informally, a query automaton is a tree automaton with the attachment of a set of "marked" states that are used to model node-selection.

Definition 5 (Query NFHA). *A query NFHA is a pair $\mathcal{Q} = (\mathcal{A}, Q_m)$ where $\mathcal{A}$ is a top-down NFHA over a set of states Q, and $Q_m \subseteq Q$ is a subset of marked states. Given a query $\mathcal{Q} = (\mathcal{A}, Q_m)$ and a tree t, a set S of positions in t is an answer to $\mathcal{Q}$ for t if there exists an $\mathcal{A}$-derivation r for t such that S is the set of positions of all nodes in r that are in $\mathcal{Q}_m$.*

Example 2 (Query NFHA) Consider the tree query $\mathcal{Q}^p = (\mathcal{A}^p, Q_m^p)$, which applied on trees from $L(\mathcal{A}^b)$, selects the node labeled *body* that is a child node of the tree root. Intuitively, this query should return an empty set of positions S for the trees t_0 and t_2, and a single position when applied on t_1 and t_3. This query is defined as follows:

— $Q^p = \{q_0, q_1, q_2, q_\top\}$, $Q_0^p = \{q_0, q_1\}$, $\Sigma^p = \Sigma^b$, $Q_m^p = \{q_3\}$,
— $\Delta^p = \{q_0 \to request(q_\top, q_2), q_1 \to response(q_\top, q_2), q_2 \to body(q_3)$, $q_3 \to \beta(q_\beta)\}$, $q_\top$ is a state which accepts all trees.

4.2 Tree Automata Composition

We now present a method for composing two hedge automata $\mathcal{A}^b$ and $\mathcal{A}^e$ based on the composition rules defined using a query NFHA $\mathcal{Q}$. The resulting automaton $\mathcal{A}$ recognizes a tree language corresponding to the substitution defined by $\mathcal{Q}$.

Let $\mathcal{Q} = (\mathcal{A}^q, Q_m)$ be a query NFHA, where $\mathcal{A}^q = (Q^q, \Sigma^b, Q_0^q, \Delta^q)$.

Given two trees t^b (base tree) and t^e (extension tree), we note $t^b[\mathcal{Q} \leftarrow t^e]$ the tree obtained by substituting t^e in t^b at every position that belongs to an answer to $\mathcal{Q}$ for t^b.

Given two sets of trees T^b and T^e, we note $T^b[\mathcal{Q} \leftarrow T^e]$ the set of trees of the form $t^b[\mathcal{Q} \leftarrow t^e]$ where $t^b \in T^b$ and $t^e \in T^e$.

Since the composition performs the intersection between the base automaton and the query, we can suppose without loss of generality that the base automaton and the query share the same alphabet. Furthermore, it is worth noticing that the query can restrict the language recognized by the base automaton. However, in practice, we consider mostly queries that are expressed using XPath: such a query accepts all trees, even if the XPath query does not select any node in some of these trees.

We consider the following core XPath language:

```
path ::= '/' relative-path
relative-path ::= step[pred] | step[pred] '/' relative-path
                | step[pred] '//' relative-path
step ::= label | '*'
pred ::= path | not(path) | pred and pred | pred or pred | true
```

We restrict `pred` to predicates that can be recognized by Boolean combinations of paths (with the usual set-to-Boolean interpretation: a path is true if and only if it matches at least one node). This ensures that these predicates can be recognized by hedge automata. Indeed, the transformation from an XPath to a query automaton is straightforward, and it is done inductively over the structure of the path. The most relevant construction is `step[pred] '/' relative-path`: given the query automata A_P (with initial state Q_P) for `pred` and A_R for `relative-path`, a new accepting state q_0 is introduced with the transition $q_0 \to step(q_\top * q_P q_\top *)$ and the resulting automaton is intersected with A_R. Resulting automata are completed such that they accept all trees, even in the case that no node is selected.

For an arbitrary tree transformation, type inference may not have a solution [12]. It is thus important to prove that for the *substitution* class of tree transformations, which we defined in Section 3, all resulting AST languages are recognizable:

Proposition 1. *Given a query $\mathcal{Q}$ and two finite hedge automata:*
$\mathcal{A}^b = (Q^b, \Sigma^b, Q_0^b, \Delta^b)$ and $\mathcal{A}^e = (Q^e, \Sigma^e, Q_0^e, \Delta^e)$,
the language $L(\mathcal{A}^b)[\mathcal{Q} \leftarrow L(\mathcal{A}^e)]$ is recognizable by a finite hedge automaton $\mathcal{A}$.

Algorithm 1 Tree automata composition

```
1: procedure COMPOSE(A^b, A^e, Q)
2:     A = (Q, Σ, Q_0, Δ);
3:     Q ← Q^e; Σ ← Σ^b ∪ Σ^e; Q_0 ← Q_0^q; Δ ← Δ^e; S ← Q_0^b × Q_0^q
4:     while S ≠ ∅ do
5:         (b, q) ∈ S; S ← S \ {(b, q)}
6:         if q ∉ Q_m^q then
7:             for all b → a(R^b) ∈ Δ^b do
8:                 for all q → a(R^q) ∈ Δ^q do
9:                     R ← {(q_1, q'_1), (q_2, q'_2), ..., (q_n, q'_n) | ∃n, q_1...q_n ∈ R^b, q'_1...q'_n ∈ R^q}
10:                    Δ ← Δ ∪ {(b, q) → a(R)}
11:                    S' ← {(b', q')|(b', q') occurs in R}
12:                    S ← S ∪ (S' \ Q)
13:                    Q ← Q ∪ S'
14:                end for
15:                if q = q_⊤ then
16:                    Δ ← Δ ∪ {(b, q) → a(R^b)}
17:                    S' ← {(b', q_⊤)|b' occurs in R}
18:                    S ← S ∪ (S' \ Q)
19:                    Q ← Q ∪ S'
20:                end if
21:            end for
22:        else
23:            for all b → a(q_f) ∈ Δ^b, q_f ∈ Q^b do
24:                for all q_0 ∈ Q_0^e do
25:                    Δ ← Δ ∪ {(b, q) → a(q_0)}
26:                end for
27:            end for
28:        end if
29:    end while
30:    return A
31: end procedure
```

Proof. It suffices to consider $\mathcal{A} = ((Q^b \times Q^q) \cup Q^e, \Sigma^b \cup \Sigma^e, Q_0^q, \Delta^e \cup \Delta)$, where Δ contains all the transitions rules of the form $(b, q) \to a(R)$ when:

— either $b \to a(R^b) \in \Delta^b$ and $q \to a(R^q) \in \Delta^q$ and R is the set $(q_1, q_1^q) \dots (q_n, q_n^q)$ when $q_1 \dots q_n \in R^b$ and $q_1^q \dots q_n^q \in R^q$.
— or $q \in Q_m$ and $b \to a'(q_f) \in \Delta^b$ and there exists $q_0 \in Q_0^e$ such that $q_0 \to a(R) \in \Delta^e$.

R is regular since R is recognized by the product of the automata that recognize respectively R^b and R^q. Based on this result, in Algorithm 1 we provide the procedure to generate $\mathcal{A}$. Next, we provide a proof that the algorithm is guaranteed to terminate with an answer for any valid input.

Proposition 2. *The Algorithm 1 terminates for all valid inputs.*

Proof. Let $\alpha \in \mathbb{N} \cup \{\omega\}$ be the number of loop iterations within the while loop between Lines 5 and 31 (possibly ω in case of non-termination). For every $i < \alpha$, let U_i be the value of $S \cup (Q^b \times Q^q) \setminus Q$ at Line 5 at the ith loop iteration. The loop satisfies $U_{i+1} \subsetneq U_i$ for every i such that $i + 1 < \alpha$. Therefore $(U_i)_{i<\alpha}$ is a sequence of strictly decreasing finite sets, thus, necessarily, $\alpha \neq \omega$.
Complexity. The size of the resulting automaton is $\mathbf{O}(|Q^e| + \left|Q^b\right| \times |Q^q|)$ and the running time is linear in the size of the output. The worst case is reached

for a family of pairs of automata $(\mathcal{A}_i^b, \mathcal{Q}_i)_i$ where, for every pair of automata $(\mathcal{A}_i^b, \mathcal{Q}_i)$, every pair of states is reachable during synchronous exploration of $\mathcal{A}_i^b$ and $\mathcal{Q}_i$.

5 Assessment

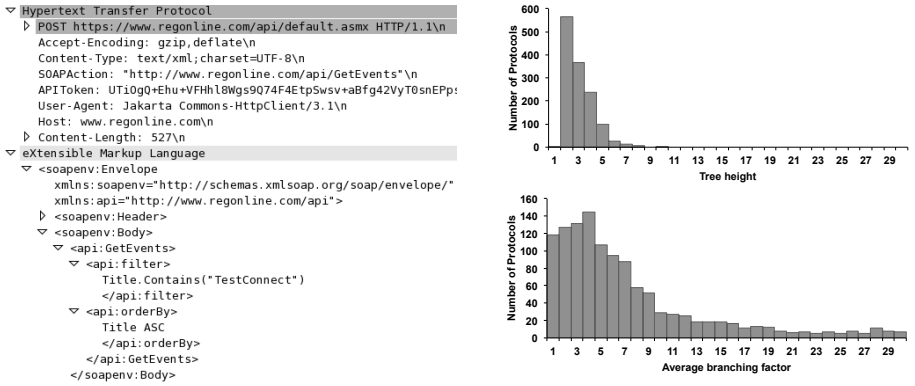
We have implemented a prototype of the proposed approach to the systematic composition of message translators, in the form of a Java library, which is available as open-source [3]. The library implements both the translator composition mechanism presented in Section 3, as well as the type inference algorithm introduced in Section 4. The purpose of this implementation is to be integrated in systems requiring high adaptability to new protocol stacks. Such systems include Enterprise Service Buses, Firewalls, Network Analyzers, etc.

While the underlying source code closely follows the formal mechanisms (such as tree automata) and algorithms presented in the paper, we further concerned ourselves with making this library usable for non-expert developers by adhering to well-established standards. Specifically, the following abstractions are concretized, as follows: (i) AST types which are internally modeled as top-down NFHAs, are transformed both on the input and output to RelaxNG (`http://relaxng.org/`) or XSD (`www.w3.org/TR/xmlschema-1/`) schema documents, and (ii) AST query inputs, which we model as query NFHAs, are to be provided using a subset of the XPath query language (`www.w3.org/TR/xpath/`).

Translator support. A prerequisite of any composition is the existence of individual translators for each protocol. In our current implementation, we integrated translators for common middleware protocols like HTTP and SOAP, as well as generic translators for extensible formats such as XML and JSON. It is important to mention that while SOAP message formats are XML-based, they are more restrictive with respect to the messages accepted, and SOAP translators also produce more compact ASTs for the same messages. It is thus interesting to have protocol-specific translators, even when the protocol itself uses an extensible data format. Other translators may be easily integrated, although the creation of associated ASTs and AST data-schemas is currently hardcoded. To overcome this limitation, we are working on a solution that will automatically inspect third-party translators using reflection and generate the two according to the data-model used by the translators.

Translator composition in Wireshark. To better asses the utility of our approach, we discuss our contributions in relation to the well-known open-source packet analyzer software Wireshark (`http://www.wireshark.org/docs/dfref/`). The role of Wireshark is to capture network packets, to parse their content and to present the information to the user in a structured format for analysis. Figure 4a depicts the representation of a HTTP/SOAP message in

[3] The project's Git repository can be checked out through anonymous access using a GIT client: `git clone https://gforge.inria.fr/git/iconnect/iconnect.git` (sourcecode located under the subproject `mtc`). Additionally, the repository can be browsed via the Git Repository Browser using the same URL. The `mtc` project is located under `projects/iconnect/iconnect.git/tree/mtc/`



(a) View of a message AST

(b) AST complexity

Fig. 4: Use of translators (aka packet dissectors) in Wireshark

the Wireshark graphical user interface. Wireshark provides two mechanisms for composing translators (which they call dissectors). The first involves writing "glue-code" as an extension of an already existing dissector implemented in the C language. A more advanced solution (postdissectors and chained dissectors), which is similar to our composition approach uses the scripting language Lua (http://wiki.wireshark.org/Lua/Dissectors) to define compositions. However, postdissectors have to be implemented in Lua, thus eliminating the possibility of using already compiled, third-party, translators. Unlike the substitution-based composition that we introduce for ASTs, Wireshark uses a much simpler approach where disjoint trees are concatenated in the result. This can be observed in Figure 4a for the trees *Hypertext Transfer Protocol* and *eXtensible Markup Language*. In Wireshark, message ASTs do not have an associated data-schema, meaning that neither individual ASTs, nor ASTs resulting from a dissector chaining can be validated or inspected before run-time. Thus, the benefit of using our composition approach in a Packet Analysis software like Wireshark, enables: (i) composition/integration of third-party translators with already existing translators, and (ii) run-time validation and static-analysis/inspection of AST data types. The second is particularly beneficial when resulting data have to be further processed (e.g., data mining and machine learning) and stored (e.g., in a relational data base) rather than simply presented to the user.

Development effort. Our entire approach is based on the assumption that developers are able to define XPath queries on message ASTs, in order to identify positions where data corresponding to composed protocols is located. In this case, it is important to estimate if this operation is effortless for developers in the general case. To this end, we conducted an analysis of AST type complexity on the 1317 protocols supported by Wireshark, focusing on two aspects that may influence the effort required to specify queries: tree height and branching factor. Intuitively, the tree height is proportional to the length of a query, when using only child axis (denoted `'/'`), and can prove complex to write in case of deep trees. Further, high branching factors (i.e., the average number of children

at each node) also make queries more complex with respect to horizontal exploration, based on node order and sibling axis. As we show in Figure 4b, both parameters are rather low in general, with the most frequent tree height being of value 2, and the most most frequent branching factor of 4. The parameters above are estimated based on the Display Filter Reference of Wireshark. Display Filters in Wireshark are quite similar to AST queries, although they are used for filtering network packets based on a predicate, rather than composing message translators. We note however, that this is only an empirical estimation knowing that the Wireshark Display Filter Reference only includes fields which are relevant for filtering, and that the hierarchical nature of message fields was deduced from the structure of field names. Furthermore, in the Display Filter Reference there is no notion of optional and mandatory fields. In the absence of this information, in the above, we considered that all fields are required. For this reason, we had to manually filter a small number of protocols which define an extensive number of optional fields (e.g., 1634 fields for the Financial Information eXchange Protocol –FIX–).

As a conclusion, we argue that the approach introduced in this paper enables developers to design composite translators seamlessly as opposed to implementing hand-coded adapters. This statement is supported by the empirical evaluation above showing that, in the general case, the XPath queries that must be provided by the developers have a low complexity.

6 Conclusions

In this paper, we presented a method for composing message translators for complex protocols stacks by reusing already exiting translator components. For systems like Packet Analyzers, Firewalls, Enterprise Service Buses, etc., the reuse of third-party translators is critical since they must constantly evolve to support an increasing number of protocol stacks. The composition approach that we introduced in this paper functions as a purely "black-box" mechanism, thus allowing the use of third-party parsers and message serializers independently of the parsing algorithm they use internally, or the method by which they were implemented/generated. Our solution goes beyond the problem of translator composition by inferring AST data-schemas relative to translator compositions. This feature allows newly generated translators to be seamlessly (or even automatically) integrated with existing systems. On a more general note, the provided inference method solves the type inference problem for the substitution class of tree compositions. This contribution has a wider domain of applications beyond the specific scope of this paper, such as the inference of XML schemas for XSLT transformations.

We implemented a prototype of the approach, which is released as open-source, to showcase its benefit in reducing development time by enabling seamless integration of message translators as reusable software components. As a part of our future work, we intend to integrate the aforementioned prototype with an Enterprise Service Bus. This will in particular allow us to further assess the benefits of the proposed message translator composition in real-life applications.

Acknowledgements. This work has been partly supported by an ANRT CIFRE grant. The authors gratefully acknowledge the suggestions and advice from Dr. Roberto Speicys Cardoso, Dr. Ajay Kattepur and Eng. Georgios Mathioudakis.

References

1. H. Basten. Ambiguity detection methods for context-free grammars. *Master's thesis, Universiteit van Amsterdam*, 2007.
2. V. Benzaken, G. Castagna, and A. Frisch. Cduce: An xml-centric general-purpose language. In *Proc. of ACM SIGPLAN ICFP '03*, 2003.
3. G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In *Middleware*. 2011.
4. L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In *DBPL*, 1994.
5. G. Castagna, H. Im, K. Nguyen, and V. Benzaken. A core calculus for XQuery 3.0. `http://www.pps.univ-paris-diderot.fr/~gc/papers/xqueryduce.pdf`, 2013.
6. D. Chappell. *Enterprise service bus*. O'reilly Media, 2004.
7. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. `http://tata.gforge.inria.fr/`, 2007. release October, 12th 2007.
8. S. DeRose and J. Clark. XML Path Language (XPath) Version 1.0. W3C recommendation, W3C, 1999.
9. M. Gallaher, A. O'Connor, J. Dettbarn, and L. Gilday. *Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry*. U.S. Department of Commerce, Technology Administration, NIST, 2004.
10. T. Joachims. Text categorization with suport vector machines: Learning with many relevant features. In *ECML*, 1998.
11. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next Generation Learnlib. In *TACAS*, 2011.
12. T. Milo and D. Suciu. Type inference for queries on semistructured data. In *PODS*, 1999.
13. A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. *KU Leuven, CW Reports vol:CW491.*, 2008.
14. M. Murata. Hedge automata: a formal model for XML schemata, 1999. `http://www.xml.gr.jp/relax/hedge_nice.html`.
15. M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM TOIT*, 2005.
16. V. Narayanan. Why I quit writing internet standards, 2014. `http://gigaom.com/2014/04/12/why-i-quit-writing-internet-standards/`.
17. Y. Papakonstantinou and V. Vianu. DTD Inference for Views of XML Data. In *PODS*, 2000.
18. T. Schwentick. Automata for XML—A Survey. *JCSS*, 2007.
19. A. C. Schwerdfeger and E. R. Van Wyk. Verifiable Composition of Deterministic Grammars. *PLDI*, 2009.
20. A. C. Schwerdfeger and E. R. Van Wyk. Verifiable Parse Table Composition for Deterministic Parsing. In *SLE*, 2010.
21. M. Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. *ACM SIGSOFT*, 1995.
22. S. D. Swierstra. Combinator parsers - from toys to tools. *ENTCS*, 41, 2000.
23. A. Tozawa. Towards Static Type Checking for XSLT. In *ACM DocEng*, 2001.