Abbreviated Path Expressions With Iterated Wild Cards: WYSIWYG Semantics and Efficient Implementation

Ahmed Abdelmeged
College of Computer &
Information Science
Northeastern University
mohsen@ccs.neu.edu

Karl Lieberherr
College of Computer &
Information Science
Northeastern University
lieber@ccs.neu.edu

ABSTRACT

Abbreviating paths with iterated wild cards is an abstraction mechanism common to Adaptive Programming (AP), eXtensible Markup Language (XML) document processing, and Aspect Oriented Programming (AOP). Recognition of abbreviated paths is used to for navigation in both AP and XML, and to decide upon advice execution in AOP. Finite state automata have been used for efficient recognition of abbreviated paths. In this paper, we introduce cover automata for abbreviated path recognition. Cover automata have significantly lower state complexity than automata used in previous approaches. One contribution of this paper is an algorithm for constructing a cover automaton for abbreviated path recognition. We also prove the correctness of our algorithm. A second contribution of this paper is a succinct formal semantics for abbreviated paths based on regular language theory which has greatly simplified our proofs.

1. INTRODUCTION

Path expressions are used to specify a set of paths pertaining to some task at hand. Path expressions are common to Object Oriented Programming (OOP), Adaptive Programming (AP), eXtensible Markup Language (XML) document processing, and Aspect Oriented Programming (AOP). In OOP, path expressions are used to retrieve information from object graphs. In AP, strategies are a form of path expression used to navigation of object graphs. In XML document processing, XPath expressions are used to specify elements in XML documents for both retrieval and update. Finally, In AOP, pointcut designators are used to specify certain join points during the course of program execution.

Abbreviating path expressions is an abstraction mechanism that allows developers to write generic programs by abstracting over irrelevant structural details. Abbreviated path expression formalisms fall into two categories: *explicit* and *implicit*. Explicit formalisms provide developers with wild cards to replace the abbreviated path components. Wild cards can also be iterated to replace multiple consecutive abbreviated path components. Examples of explicit formalisms is XPath where developers can use "//" as form of iterated wild card and the AspectJ point cut language that provides the "cflow" construct as a form of iterated wild cards. The ".*" in regular expressions is a third example of iterated wild cards.

Implicit formalisms do not provide developers with wild cards. Instead, all paths are translated into an explicit form by inserting wild cards into certain places defined by what we call an *expansion semantics*. Implicit formalisms save the developers from writing too many wild cards to make their methods flexible. An example of implicit formalisms is the regular-expression-like strategy language in AP.

Naive recognition of path expressions with iterated wild cards is inefficient. For example, retrieval of nodes specified by the XPath expression ".//para" involves a traversal of the entire XML tree and might visit a large number of nodes that can never lead to a "para" element. Schema information can be used to optimize the execution of path expressions with wild cards so that it visits nodes from which a "para" element is reachable. Several constructions [7, 4, 11] exist that are based on the idea of intersecting two automata one representing paths in the query and another representing paths in the schema. The number of states in the intersection automata is proportional to the product of number of states in both automata.

Observing the fact that at runtime the intersection automata is used to check paths in an document that conform to the schema. A cover automata for the intersection can be used. A cover automata can have more paths than the intersection automata has as long as these extra paths are illegal according to the schema. Cover automata can have a significantly lower state complexity than intersection automata. One contribution of this paper is an algorithm for constructing a cover automaton for abbreviated path recognition. We also prove the correctness of our algorithm.

In AP, the classical interpretation of wild cards as place holders for "anything" leads to modularity and ambiguity problems. In this paper, we argue that a slightly restricted interpretation of wild cards, called WYSIWYG, can solve both problems. Furthermore, it can improve the efficiency of recognizing abbreviated paths with wild cards. We give a succinct formal semantics for abbreviated path expressions regular language theory which has greatly simplified our proofs.

The rest of this paper is organized as follows: In section 2, we introduce our notation. In section 3, we introduce the WYSIWYG interpretation of wild cards. In section 4, we show that it is possible to improve the efficiency of path recognition. In section 5, we give a construction of an efficient recognizer for abbreviated path expressions with iterated wild cards, In section 6, we discuss some of the related work. In section 7, we conclude this paper. It is worth mentioning that sections 3 and 4 are independent of each other and can be read separately.

2. NOTATION

We use

- uppercase Greek letters to denote alphabets (e.g. Σ_{C}),
- lowercase Greek letters to denote strings,
- uppercase Latin letters to denote regular languages (e.g. C),
- lowercase Latin letters to denote symbols (e.g. a) as well as functions (e.g. $meta$),
- Σ^* to denote the *free monoid* on an alphabet Σ,
- f^* to denote a homomorphism $f^* : \Sigma_A^* \to \Sigma_B^*$ constructed by extending the function $f : \Sigma_A \to \Sigma_B^*$ the usual way,
- $\mathbb{C}$ to denote the state complexity of a regular language C. The state complexity of a regular language is the number of states in the minimal deterministic finite automaton that accepts it,
- R° to denote the prefix closure of a regular language R. Formally, $R^\circ = \{\omega \mid \exists \sigma \in R : \omega \sqsubseteq \sigma\}$.
- $\mathcal{P}(S)$ to denote the power set of some set S.

3. STATIC SEMANTICS OF ABBREVIATED PATHS WITH ITERATED WILD CARDS

Given:

- An alphabet Σ_{C}. Words over Σ_{C} are called concrete paths.
- A set of abbreviated paths $A \subseteq (\Sigma_{\mathsf{A}} \cup \diamond)^*$ where $\Sigma_{\mathsf{A}} \subseteq \Sigma_{\mathsf{C}}$ and $\diamond \notin \Sigma_{\mathsf{C}}$ is a distinguished wild card symbol.
- All occurrences of $\diamond$ are iterated (i.e. $\diamond$ only shows under the Kleene star). Formally, $\forall \alpha, \beta \in (\Sigma_{\mathsf{A}} \cup \diamond)^* : \alpha \cdot \diamond \cdot \beta \in A \Rightarrow \alpha \cdot \diamond^* \cdot \beta \subseteq A$.

The meaning of a set of abbreviated paths A with respect to a set of concrete paths C according to the WYSIWYG semantics, denoted WWG(A, C), is a subset of paths in C obtainable from A by replacing wild cards in some path in A with symbols from $\Sigma_{\mathsf{C}} \backslash \Sigma_{\mathsf{A}}$. Given a concrete path ω, there can be at most one corresponding abbreviated path α. Furthermore, α can be obtained from ω by replacing all occurrences of symbols not in Σ_{A} in ω by wild cards. This observation enables us to have the following succinct formal definition for WWG(A, C):
WWG(A, C) $= C \cap \{\alpha \in \Sigma_{\mathsf{C}}^* \mid f^*(\alpha) \in A\}$, Where:

$$f(a) = \begin{cases} a & , a \in \Sigma_{\mathsf{A}} , \\ \diamond & , otherwise . \end{cases}$$

Throughout the rest of this section we shall contrast WWG(A, C) to the classic interpretaion CLASSIC(C, A) of abbreviated paths in which wild cards can be replaced by symbols from Σ_{C}.

3.1 Modularity

The purpose of using abbreviated paths in module M to refer to some structure defined in another module N is to lower the coupling between M and N. In the context of AP, abbreviated paths mentioned in a method select a subset of paths in the class graph. The CLASSIC interpretation of a set A of abbreviated paths often contains more paths than it *should*. And consequently, selects more paths in the class graph than it *should*. These extra paths are referred to in the literature as *surprise paths*. The solution to *surprise paths* is to identify and *bypass* those paths. This solution increases the coupling between methods and the class graph.

To illustrate this issue, consider the "Bus Route Class Graph" shown in Figure 1. Suppose that we are using the abbreviated path expression $A = BusRoute \cdot \diamond^* \cdot Passenger$ to select paths in the class graph. According to both the CLASSIC and the WYSIWYG semantics, the set of selected paths is $BusRoute \cdot LoB \cdot LoB^* \cdot Bus \cdot LoP \cdot LoP^* \cdot Passenger$. Now assume that $Passes$ were added to the class graph as indicated in Figure 2. According to the WYSIWYG semantics, the set of selected paths does not change. However, according to the CLASSIC semantics, the set of selected paths becomes $BusRoute \cdot LoB \cdot LoB^* \cdot Bus \cdot LoP \cdot LoP^* \cdot Passenger \cdot (Pass \cdot BusRoute \cdot LoB \cdot LoB^* \cdot Bus \cdot LoP \cdot LoP^* \cdot Passenger)^*$. This illustrates the increased coupling. Furthermore, to bypass the extra paths, we need to mention the "Noise" class $Passenger$ in our abbreviated path expression, increasing the coupling even further.

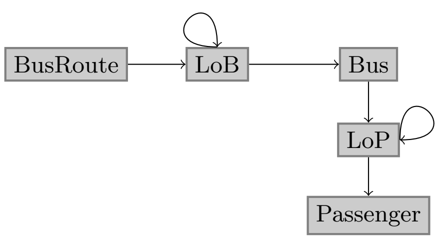

Figure 1: Bus Route Class Graph

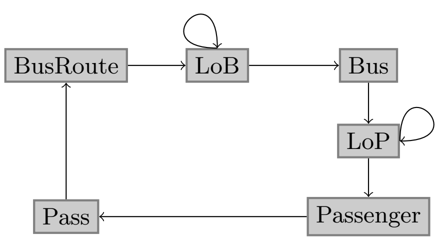

Figure 2: Evolved Bus Route Class Graph

3.2 Ambiguity

Ambiguity is not a problem for recognition. It becomes a problem when events during the recognition process are observed. In AP, we associate behavior with paths in an abbreviated path expression. Therefore, confusion can occur when one concrete path can match more than one abbreviated path. As mentioned earlier, with the WYSIWYG semantics, there can be at most one abbreviated path corresponding to some concrete path. With the CLASSIC semantics, ambiguity can occur. For example, consider the set $A = a \cdot \diamond^* \cdot b \cdot \diamond^* \cdot d \ \cup \ a \cdot \diamond^* \cdot c \cdot \diamond^* \cdot d$, and the concrete path $\omega = a \cdot b \cdot c \cdot d$. According to the CLASSIC semantics, ω matches both $a \cdot b \cdot \diamond \cdot d$, and $a \cdot \diamond \cdot c \cdot d$. According to WYSIWYG semantics, ω matches neither.

3.3 Efficiency

Another, important property of the set WWG(A, C) is that it has the same state complexity as the set A, which, as we shall see later, is directly related to the efficiency of its recognition.

THEOREM 3.1 (EFFICIENCY). *Let* $W = WWG(A, C)$, $\mathbb{W} \leq \mathbb{A} * \mathbb{C}$.

PROOF. Let $AA = \langle Q, \Sigma_{\mathsf{A}} \cup \diamond, \delta, q_0, F\rangle$ be the minimal DFA that recognizes A. From Automata Theory, the DFA $EXP(AA, \Sigma_{\mathsf{C}}) = \langle Q, \Sigma_{\mathsf{C}}, \gamma, q_0, F\rangle$, where $\gamma(q_i, a) = \delta(q_i, f(a))$, recognizes $EXP \Sigma_{\mathsf{C}} A$. Furthermore, $EXP(AA, \Sigma_{\mathsf{C}})$ has the same number of states as AA which is $\mathbb{A}$. let CA be the minimal DFA that recognizes C. By definition, C has $\mathbb{C}$ states. Therefore, by definition of $WWGAC$, the minimal DFA for recognizing it has at most $mathbbA * \mathbb{C}$ states [13]. $\square$

A similar construction for CLASSIC(A, C) results in a nondeterministic finite automaton. Therefore, the state complexity of CLASSIC(A, C) can be exponentially larger than WWG(A, C). An example illustrating this exponential complexity is given in [7].

4. DYNAMIC SEMANTICS

Given:

- A schema modeled as a language $C \subseteq \Sigma_{\mathsf{C}}^*$. We call Σ_{C} the set of classes.
- A set of specified paths modeled as another regular language $\mathrm{S} \subseteq C$. Typically, S is the meaning of a set of abbreviated paths with with respect to C.
- A set of object paths modeled as a regular language $O \subseteq \Sigma_{\mathsf{O}}^*$. We call Σ_{O} the set of objects. Typically, O is the set of all paths that were once active during some traversal of some graph defined over the set of objects.
- A function $meta : \Sigma_{\mathsf{O}} \rightarrow \Sigma_{\mathsf{C}}$ that maps objects to classes. We say that $\pi \in \Sigma_{\mathsf{O}}^*$ is an instance of $meta^*(\pi)$. We also define a function slice$(L, O) = \{\pi \in O \mid meta^*(\pi) \in L\}$ to return a subset of object paths that are instances of some path in L.
- O conforms to C, dentoted conforms(O, C). Formally, conforms(O, C) means slice(C, O) = O.

The dynamic semantics of a set of paths S with respect to an object graph O, denoted goal(S, O) is the set of object graph paths that can be legally completed to an instance of some path in S. Formally, goal(S, O) = slice(S°, O). It is worth mentioning that the state complexity of S° is the same as the state complexity ofS because a DFA recognizing S° can be constructed from a DFA recognizing S by turning every state leading to a final state into a final state.

4.1 Cover Languages

It is desirable to construct the smallest possible recognizer for S° in order to improve the overall performance at runtime. One approach to reduce its size is to construct a nondeterministic finite state recognizer. This approach was adopted in [7], and it is possible when the language S° has a minimal nondeterministic finite state recognizer that is smaller than the minimal deterministic finite state recognizer. This was the case with the classic interpretation of wild cards adopted there. However, this is not the case in general and certainly it is not the case when the WYSIWYG interpretation of wild cards is adopted.

Fortunately, there is another approach. We can rely on the fact that only legal object graph paths are going to be checked against S° at runtime, and use a cover language for S° that also contain some illegal paths. The state complexity of the cover language can be significantly lower than the original language itself. For example, if $C = \mathrm{S}$, then $C = \mathrm{S} \cap C$ meaning that all legal paths are selected. Since at runtime we are going to check only legal paths, the cover language Σ_{C}^* (whose state complexity is 1) can be used.

A cover language cover(S, C) for the language S with respect to C is formally defined as: cover$(S, C) = \{X \subseteq \Sigma_{\mathsf{C}}^* \mid X \cap C = C\}$. An automata recognizing cover(S, C) is called a cover automata for S with respect to C.

THEOREM 4.1 (CORRECTNESS). $\forall O, C, S :$ *conforms*$(O, C) \Rightarrow$ *slice*(*cover*$(S, C), O) =$ *slice*(S, O).

PROOF. slice(cover(S, C), O)
$= \{\omega \in O \mid meta^*(\omega) \in \text{cover}(S, C)\}$
$= \{\omega \in O \mid meta^*(\omega) \in \text{cover}(S, C) \cap C\}$
$= \{\omega \in O \mid meta^*(\omega) \in S\}$
$=$ slice(S, O). $\square$

5. RECOGNIZING ABBREVIATED PATH EXPRESSIONS WITH COVER AUTOMATA

Given:

- A class graph modeled as a pair $\mathsf{C} = \langle \Sigma_{\mathsf{C}}, E_{\mathsf{C}} \rangle$ where $\Sigma_{\mathsf{C}} \neq \emptyset$ is a non empty set of nodes, called classes, and $E_{\mathsf{C}} \subseteq \Sigma_{\mathsf{C}} \times \Sigma_{\mathsf{C}}$ is a set of edges. Let C be the regular language of all paths in C. By its definition, C has the following two properties:
  - $C^\circ = C$ and
  - $\forall \alpha, \beta \in \Sigma_{\mathsf{C}}^*, x \in \Sigma_{\mathsf{C}} : \alpha \cdot x \in C \wedge x \cdot \beta \in C \Rightarrow \alpha \cdot x \cdot \beta \in C$.
- An Automaton $S = \langle Q, \Sigma_{\mathsf{S}} \cup \diamond, \delta, q_0, F\rangle$ representing a set of abbreviated paths. We require S to have the following four properties:
  - S has only one stuck state denoted $q_\perp \notin F$. Formally, $\forall a \in \Sigma_{\mathsf{S}} \cup \diamond : \delta(q_\perp, a) = q_\perp$ and $\forall q_i \in Q \backslash q_\perp : \exists a \in \Sigma_{\mathsf{S}} \cup \diamond \ s.t. \ \delta(q_i, a) \neq q_\perp$.
  - All wild card symbols appear on loops. $\forall q_i : \delta(q_i, \diamond) \neq q_\perp \Rightarrow \delta(q_i, \diamond) = q_i$.
  - S is compatible with C meaning that every transition labeled with a symbol from Σ_{S} be part of some fruitful path. Formally, $\forall q_i \in Q, a \in \Sigma_{\mathsf{S}} : \delta(q_i, a) \neq q_\perp \Rightarrow \exists \beta \in \Sigma_{\mathsf{C}}^* \ s.t. \ a \cdot \beta \in C \wedge \delta^*(q_i, f^*(a \cdot \beta)) \in F$.

– $\mathcal{L}(S)$ has at least one fruitful path, $\exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; \beta \in C \wedge \delta^*(q_0, f^*(\beta)) \in F$.

We show how to construct a an automata for recognizing a cover language for
WWG$(\mathcal{L}(S), C)^\circ$. We also prove the constructed automata correct and show that the constructed automata has the same number of states as S.

$RR(S, C) \equiv \langle Q, \Sigma_{\mathsf{C}}, \eta, q_0, Q \setminus \{q_\perp\}\rangle$

where :

$$\eta(q_i, a) = \begin{cases} \delta(q_i, f(a)) & \text{if } a \in \Sigma_{\mathsf{S}} \cup \Delta_{q_i}, \\ q_\perp & \textit{otherwise}. \end{cases}$$

$$\Delta_{q_i} = \{a \in (\Sigma_{\mathsf{C}} \backslash \Sigma_{\mathsf{S}}) \mid \exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; a \cdot \beta \in C \wedge \delta^*(q_i, f^*(a \cdot \beta)) \in F\}$$

Lemma 5.1. $\forall q_i \in Q, a \in \Sigma_{\mathsf{C}} : \eta(q_i, a) \neq q_\perp \Leftrightarrow \exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; a \cdot \beta \in C \wedge \delta^*(q_i, f^*(a \cdot \beta)) \in F$.

The automaton $RR(S, C)$ gets into a stuck state iff there is no way to achieve a fruitful path, i.e., a path that is both in C is selected by S.

Proof. $\Rightarrow$ direction:
case $a \in \Sigma_{\mathsf{S}}$: Immediate, from the definition of η and the compatibility condition.
case $a \in \Delta_{q_i}$: Immediate, from the definition of Δ_{q_i}.
case otherwise: from the definition of η, $\eta(q_i, a) = q_\perp$.

$\Leftarrow$ direction:
By the definition of Δ_{q_i} and the compatibility condition, $a \in \Sigma_{\mathsf{S}} \cup \Delta_{q_i}$. Therefore, from the definition of η, $\eta(q_i, a) = \delta(q_i, f(a))$. But, $\delta(q_i, f(a)) \neq q_\perp$ because $\delta^*(q_i, f^*(a \cdot \beta)) = \delta^*(\delta(q_i, f(a)), f^*(\beta)) \in F$ and by definition of $q_\perp$, $q_\perp \notin F$ and $\forall\alpha \in \Sigma_{\mathsf{C}}^* : \delta^*(q_\perp, \alpha) = q_\perp$. □

Lemma 5.2. $\forall q_i \in Q, \alpha \in \Sigma_{\mathsf{C}}^* : \eta^*(q_i, \alpha) \neq q_\perp \Rightarrow \eta^*(q_i, \alpha) = \delta^*(q_i, f^*(\alpha))$.

Proof. Immediate, by simple induction on $|\alpha|$. □

Theorem 5.3 (correctness).
$\mathcal{L}(RR(S, C)) \in$ *cover*$(WWG(\mathcal{L}(S), C)^\circ, C)$.

Proof. We show that:

1. $\mathcal{L}(RR(S, C)) \cap C \subseteq$ WWG$(\mathcal{L}(S), C)^\circ C$.
2. WWG$(\mathcal{L}(S), C)^\circ C \subseteq \mathcal{L}(RR(S, C)) \cap C$.

1. $\mathcal{L}(RR(S, C)) \cap C \subseteq$ WWG$(\mathcal{L}(S), C)^\circ C$.
Which reduces to:
$\mathcal{L}(RR(S, C)) \cap C^\circ \subseteq (C \cap \{\omega \in \Sigma_{\mathsf{C}}^* \mid \delta^*(q_0, f^*(\omega)) \in F\})^\circ$
Which, by the defnition of prefix closure, reduces to:
$\mathcal{L}(RR(S, C)) \cap C^\circ \subseteq \{\alpha \in \Sigma_{\mathsf{C}}^* \; s.t. \; \exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; \alpha \cdot \beta \in (C \cap \{\omega \in \Sigma_{\mathsf{C}}^* \mid \delta^*(q_0, f^*(\omega)) \in F\})\}$
Which further reduces to: $\forall\alpha \in \Sigma_{\mathsf{C}}^* \; s.t. \; \eta^*(q_0, \alpha) \neq q_\perp \wedge \alpha \in C : \exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; \alpha \cdot \beta \in C \wedge \delta^*(q_0, f^*(\alpha \cdot \beta)) \in F$

We proceed by induction on $|\alpha|$:
Base case: $(|\alpha| = 0)$
$\alpha = \varepsilon$ and therefore, $\eta^*(q_0, \varepsilon) = q_0$. By the definition of S, $\exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; \beta \in C \wedge \delta^*(q_0, f^*(\beta)) \in F$. Therefore, $\alpha \cdot \beta \in C$, and $\delta^*(q_0, f^*(\alpha \cdot \beta)) \in F$.

Induction step: Let $\alpha = \omega \cdot a$
Therefore, $\eta^*(q_0, \omega \cdot a) \neq q_\perp$. But, $\eta^*(q_0, \omega \cdot a) = \eta(q_i, a)$, where $q_i = \eta^*(q_0, \omega)$. Therefore, $q_i \neq q_\perp$, and by lemma 5.2, $\delta^*(q_0, f^*(\omega)) = q_i$. Furthermore, $\eta(q_i, a) \neq q_\perp$. Therefore, by lemma 5.1, $\exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; a \cdot \beta \in C \wedge \delta^*(q_i, f^*(a \cdot \beta)) \in F$. But, $\omega \cdot a \in C$, therefore, by the definition of C, $\omega \cdot a \cdot \beta = \alpha \cdot \beta \in C$. And, $\delta^*(q_i, f^*(a \cdot \beta)) = \delta^*(\delta^*(q_0, f^*(\omega)), f^*(a \cdot \beta)) = \delta^*(q_0, f^*(\omega \cdot a \cdot \beta)) = \delta^*(q_0, f^*(\alpha \cdot \beta))$, therefore,
$\delta^*(q_0, f^*(\alpha \cdot \beta)) \in F$.

2. WWG$(\mathcal{L}(S), C)^\circ C \subseteq \mathcal{L}(RR(S, C)) \cap C$.
Which reduces to:
$(C \cap \{\omega \in \Sigma_{\mathsf{C}}^* \mid \delta^*(q_0, f^*(\omega)) \in F\})^\circ \subseteq \mathcal{L}(RR(S, C))$
Which, by the defnition of prefix closure, reduces to: $\{\alpha \in \Sigma_{\mathsf{C}}^* \; s.t. \; \exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; \alpha \cdot \beta \in (C \cap \{\omega \in \Sigma_{\mathsf{C}}^* \mid \delta^*(q_0, f^*(\omega)) \in F\})\} \subseteq \mathcal{L}(RR(S, C))$
Which further reduces to: $\forall\alpha \in \Sigma_{\mathsf{C}}^* \; s.t. \; \exists\beta \in \Sigma_{\mathsf{C}}^* \; s.t. \; \alpha \cdot \beta \in C \wedge \delta^*(q_0, f^*(\alpha \cdot \beta)) \in F : \eta^*(q_0, \alpha) \in Q \backslash \{q_\perp\}$
Which, by the definition of $RR(S, C)$, reduces to: $\forall\alpha \in C \; s.t. \; \exists\beta \in C \; s.t. \; \alpha \cdot \beta \in C \wedge \delta^*(q_0, f^*(\alpha \cdot \beta)) \in F : \eta^*(q_0, \alpha) \neq q_\perp$

We proceed by induction on $|\alpha|$:
base case: $(|\alpha| = 0)$
$\alpha = \varepsilon$ and therefore, $\eta^*(q_0, \varepsilon) = q_0 \neq q_\perp$.
induction step: Let $\alpha = \omega \cdot a$
$\eta^*(q_0, \alpha) = \eta^*(q_0, \omega \cdot a) = \eta(q_i, a)$, where $q_i = \eta^*(q_0, \omega)$. By induction hypothesis, $q_i \neq q_\perp$. By lemma 5.2, $\delta^*(q_0, f^*(\omega)) = q_i$. But, $\delta^*(q_0, f^*(\alpha \cdot \beta)) = \delta^*(q_0, f^*(\omega \cdot a \cdot \beta)) =$
$\delta^*(\delta^*(q_0, f^*(\omega)), f^*(a \cdot \beta)) = \delta^*(q_i, f^*(a \cdot \beta))$, therefore, $\delta^*(q_i, f^*(a \cdot \beta)) \in F$. And $\alpha \cdot \beta \in C$, therefore, $a \cdot \beta \in C$. By lemma 5.1, we can conclude $\eta(q_i, a) \neq q_\perp$. Hence, $\eta^*(q_0, \alpha) \neq q_\perp$. □

Theorem 5.4 (efficiency). *$RR(S, C)$ has the same number of states as S.*

Proof. Immediate from the definition of $RR(S, C)$ □

6. RELATED WORK

The WYSIWYG expansion idea was introduced from a complexity prespective in [8] under the name of pure paths. WYSIWYG semantics was further explored in [12, 1] which provide a first glimpse at the treatment provided in this paper. But the proofs are long. The solution provided in this paper is clearly better and provides a nice demonstration of the power of regular languages.

Disambiguation techniques for matching a single string against a regular expression are discussed in [5, 2]. Mendelzon and Wood analyzed the complexity of finding regular paths in graphs [9]. They showed that finding simple regular paths in a graph is NP-complete problem while finding regular paths is a polynomial-time problem. Sereni and de Moor study the static determination of cflow pointcuts in AspectJ [11]. They reason also in terms of sets of paths, and they use regular expressions as pointcut language. They model pointcut designators as automata. They do whole program analysis on the call graph of the program and try to determine whether a potential join point fits into one of the following three cases: (1) it *always* matches a cflow pointcut; (2) it *never* matches a cflow pointcut; (3) it *maybe* matches a cflow pointcut. In case (3), there is still a need to have dynamic matching code.

Automata are widely used to evaluate XPath Queries on XML documents [6, 3]. Some use schemas to speed up the processing [4] while others do not [3]. Simple, but very effective techniques, such as the Stream Index [6] are used to skip over irrelevant document parts. None of the papers in the XML literature use our idea of cover automata to significantly simplify the deterministic automata.

7. CONCLUSION

This paper brings a line of work, called the Theory of Traversals for Adaptive Programming [10, 7, 1, 12] to a natural conclusion. The paper simplifies the model to its essence which allows a very elegant derivation of provably correct algorithms for implementing basic tools useful for numerous applications, e.g., XML, AOP and AP. Correctness arguments that used to fill many pages are reduced to just a few lines.

8. REFERENCES

[1] A. Abdelmeged, T. Skotiniotis, and K. Lieberherr. Navigating Object Graphs Using Incomplete Meta-Information. Technical Report NU-CCIS-10-2, CCIS/PRL, Northeastern University, Boston, March 2010.

[2] C. Brabrand and J. G. Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. 2010.

[3] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[4] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 14–23, Washington, DC, USA, 1998. IEEE Computer Society.

[5] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. of ICALPÕ04*, pages 618–629, 2004.

[6] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[7] K. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[8] K. J. Lieberherr and B. Patt-Shamir. The refinement relation of graph-based generic programs. In M. Jazayeri, R. Loos, and D. Musser, editors, *1998 Schloss Dagstuhl Workshop on Generic Programming*, pages 40–52. Springer, 2000. LNCS 1766.

[9] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. In *VLDB*, pages 185–193, 1989.

[10] J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, Mar. 1995.

[11] D. Sereni and O. de Moor. Static analysis of aspects. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 30–39, New York, NY, USA, 2003. ACM.

[12] T. Skotiniotis. *Modular Adaptive Programming.* PhD thesis, Northeastern University, 2010. 190 pages.

[13] S. Yu. State complexity of regular languages. *J. Autom. Lang. Comb.*, 6(2):221–234, 2001.