Universiteit Leiden
Opleiding Informatica

Partial derivatives

for KAT expressions

Name: Jennifer Jochems
Date: 14/06/2014

1st supervisor: Dr. Marcello Bonsangue
2nd supervisor: Dr. Hendrik Jan Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Table of Contents

Abstract

We develop a method for constructing non-deterministic finite automata (NFA) for Kleene algebra with tests (KAT) expressions for use in program verification. This method uses partial derivatives to construct NFAs. The `first` set of an expression determines which derivatives to compute, thus reducing the computation steps. Our NFAs are the first finite automata to have the same complexity as their Kleene algebra counterparts. To check the constructed automata for equivalence, we use a bisimulation up-to congruence method. This allows for checking equivalence 'on-the-fly' without the need of prior determinization.

1 Introduction

Program verification is a crucial part of software development. One way of verifying programs is by modeling programs as algebraic expressions. A Kleene algebra (KA) of regular expressions is often used in this context. The real life applications of KA in program verification are very limited, though. Conditional statements and loops cannot be modeled using solely Kleene algebra. Because conditional statements are some of the most powerful and frequently used programming constructs, KA needs to be enhanced in order to be used in program verification.

Therefore, KA has been extended to Kleene algebra with tests (KAT) [1] to include a Boolean subalgebra. Section 2 formalizes the notion of KAT. The Boolean algebra allows for the modeling of tests and thus conditional statements and loops. This makes KAT suited for verifying a wide range of programs. In fact, KAT has successfully been used in basic safety analysis, source-to-source program transformation and concurrency control [2]. An advantage of using KA over e.g. propositional logic or Hoare logic, is that algebraic expressions are easy to manipulate and KA allows for the construction of finite state automata. Finite automata can easily be used to check the equivalence between two KA expressions. Automata for KAT expressions can be constructed in a similar way. These automata can consequently be used in program verification.

As for KA, KAT finite state automata come in three types: deterministic (DFA), non-deterministic (NFA) and non-deterministic with Λ-transitions (NFA-Λ). DFAs have the advantage that equivalence checking is relatively straight-forward. At the same time, the language of a DFA is hard to read and the size of a DFA explodes with many transitions going to a trap state. The latter is especially true for KAT expressions as demonstrated in Section 3. For NFA-Λ, the Λ-transitions make the language hard to read. Therefore, we choose to work with NFAs in this thesis. Section 3 describes the construction method for KAT NFAs in detail.

The main aim of this thesis is to efficiently construct NFAs for KAT expressions and efficiently compute the equivalence relation between two such NFAs. Thus, efficiently checking whether two programs are equivalent. In order to evaluate the efficiency, Section 3.3 investigates the complexity of the proposed construction method. In Section 4 we present an algorithm for computing the equivalence relation between two NFAs for KAT. The algorithm uses a bisimulation up-to congruence method inspired by Pous et al. 2013 [3] to limit the number of computation steps. Section 5 provides an elaborate example of how the results presented in this thesis can be applied. Section 6 compares our results to earlier obtained results. This section also evaluates the efficiency and usability of our methods.

2 Preliminaries

2.1 Algebraic structures

In order to fully appreciate the algebras discussed in this thesis, one has to be familiar with a number of algebraic structures. This mathematical background serves as a foundation to learn about KAT.

A *group* is a tuple $(G, +, -, 0)$ consisting of a set G with a binary operation $+ : G \times G \to G$, a unary operation $- : G \to G$ and an identity element 0. A group satisfies the following axioms for all $a, b, c \in G$:

$$(a + b) + c = a + (b + c) \tag{1}$$
$$0 + a = a \tag{2}$$
$$a + 0 = a \tag{3}$$
$$a + -a = 0 \tag{4}$$
$$-a + a = 0 \tag{5}$$

Axiom (1) is the associativity axiom. Axioms (2) and (3) denote the existence of a unique identity element. Axioms (4) and (5) denote the existence of an inverse element. The binary operation $+$ is not

necessarily commutative. A group that is commutative, thus $\forall a, b \in G : a + b = b + a$, is called an *Abelian group*. A *monoid* is very similar to a group. In fact, each group is a monoid while the converse does not necessarily hold. Thus, $(M, +, 0)$ satisfies all but the last two axioms: elements in a monoid M need not have an inverse element in M. A monoid may also be commutative.

A *ring* is a tuple $(R, \cdot, +, -, 0, 1)$ consisting of a set R with binary operations $\cdot, + : R \times R \to R$, a unary operation $- : R \to R$ and identity elements 1 and 0, respectively. A ring is an Abelian group under addition and a monoid under multiplication. On top of that, multiplication distributes over addition. Thus, for all $a, b, c \in R$:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c) \tag{1}$$
$$(b + c) \cdot a = (b \cdot a) + (c \cdot a) \tag{2}$$

A *semiring* relates to a ring similar to how a monoid relates to a group. A semiring satisfies all ring axioms except for the additive inverse requirement. This means that it is a commutative monoid under addition, rather than an Abelian group.

A *lattice* is a tuple $(L, \cdot, +)$ consisting of a set L with binary operations $\cdot, + : L \times L \to L$. A lattice satisfies a total of eight axioms. $(L, \cdot)$ and $(L, +)$ define semilattices, see axioms (1)-(3) and (4)-(6), respectively. Axioms (7) and (8) distinguish a lattice from an arbitrary pair of semilattices. For all $a, b, c \in L$:

$$(a + b) + c = a + (b + c) \tag{1}$$
$$a + b = b + a \tag{2}$$
$$a + a = a \tag{3}$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \tag{4}$$
$$a \cdot b = b \cdot a \tag{5}$$
$$a \cdot a = a \tag{6}$$
$$a + (a \cdot b) = a \tag{7}$$
$$a \cdot (a + b) = a \tag{8}$$

A *bounded lattice* is a lattice with 0 as its bottom element and 1 as its top element such that 0 is the additive identity and 1 is multiplicative identity.

2.2 Regular expressions and regular languages

Let the alphabet Σ be a non-empty, finite set of symbols. Then Σ^* denotes the set of all words over Σ. A language over Σ is a subset of Σ^*. Note that * is the usual Kleene star. Thus:

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$$

Regular expressions over the alphabet Σ and the regular languages they denote are inductively defined. The base cases are as follows:

- 0 is a regular expression denoting the language $L(0) = \emptyset$
- 1 is a regular expression denoting the language $L(1) = \{\epsilon\}$
- $\forall a \in \Sigma : a$ is a regular expression denoting the language $L(a) = \{a\}$

Now let F and G be regular expressions denoting the language $L(F)$ and $L(G)$, respectively. Then by induction we define:

- $F + G$ is a regular expression denoting the language $L(F + G) = L(F) \cup L(G)$
- $F \cdot G$ is a regular expression denoting the language $L(F \cdot G) = L(F) \cdot L(G)$
- F^* is a regular expression denoting the language $L(F^*) = (L(F))^*$

No other expression is a regular expression. Two regular expressions are considered equal if the languages they denote are equal. As usual, $\cdot$ takes precedence over $+$.

2.3 Kleene algebra

A Kleene algebra is a tuple $(K, \cdot, +, ^*, 0, 1)$ consisting of a set K with binary functions $\cdot, + : K \times K \to K$, unary function $^* : K \to K$ and identity elements 1 and 0, respectively. For all $a, b, c, x \in K$, a Kleene algebra satisfies the following axioms [4]:

$$a + (b + c) = (a + b) + c \tag{1}$$
$$a + b = b + a \tag{2}$$
$$a + 0 = a \tag{3}$$
$$a + a = a \tag{4}$$
$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \tag{5}$$
$$1 \cdot a = a \tag{6}$$
$$a \cdot 1 = a \tag{7}$$
$$a \cdot (b + c) = a \cdot b + a \cdot c \tag{8}$$
$$(a + b) \cdot c = a \cdot c + b \cdot c \tag{9}$$
$$0 \cdot a = 0 \tag{10}$$
$$a \cdot 0 = 0 \tag{11}$$
$$1 + a \cdot a^* \leq a^* \tag{12}$$
$$1 + a^* \cdot a \leq a^* \tag{13}$$
$$b + a \cdot x \leq x \Rightarrow a^* \cdot b \leq x \tag{14}$$
$$b + x \cdot a \leq x \Rightarrow b \cdot a^* \leq x \tag{15}$$

where $\leq$ denotes the natural partial order on K:

$$a \leq b \Leftrightarrow a + b = b.$$

It follows from axioms (1)-(4) that $(K, +, 0)$ is an idempotent commutative monoid. Axioms (5)-(7) demonstrate that $(K, \cdot, 1)$ is a monoid. Then, it follows from axioms (1)-(11) that $(K, \cdot, +, 0, 1)$ is an idempotent semiring. The remaining axioms define the behavior of the Kleene star *.

In this thesis we consider a specific example of a Kleene algebra. It can be shown that the set of regular expressions as defined in Section 2.2 satisfies the axioms above and therefore is a Kleene algebra. This is also true for the set of regular languages. A regular expression over the alphabet Σ is a term of the Kleene algebra with generator Σ. This is the specific Kleene algebra we work with.

2.4 Boolean algebra

A Boolean algebra is a tuple $(B, \cdot, +, ^-, 0, 1)$ consisting of a set B with binary functions $\cdot, + : B \times B \to B$, unary function $^-$: $B \to B$ and identity elements 1 and 0, respectively. For all $a, b, c \in B$, a Boolean algebra satisfies the following axioms:

$$a + (b + c) = (a + b) + c \tag{1}$$
$$a + b = b + a \tag{2}$$
$$a + 0 = a \tag{3}$$
$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \tag{4}$$
$$a \cdot b = b \cdot a \tag{5}$$
$$a \cdot 1 = a \tag{6}$$
$$a + (a \cdot b) = a \tag{7}$$
$$a \cdot (a + b) = a \tag{8}$$
$$a + \bar{a} = 1 \tag{9}$$
$$a \cdot \bar{a} = 0 \tag{10}$$
$$a + (b \cdot c) = (a + b) \cdot (a + c) \tag{11}$$
$$a \cdot (b + c) = (a \cdot b) + (a \cdot b) \tag{12}$$

It follows from axioms (1)-(8) that $(B, \cdot, +)$ is a lattice. Note that the idempotency axioms follow directly from absorption axioms (7) and (8). Axioms (3) and (6) make $(B, \cdot, +, 0, 1)$ a bounded lattice. It follows from axioms (9) and (10) that $(B, \cdot, +, \bar{}, 0, 1)$ is a complemented lattice. Ultimately, it follows from axioms (11) and (12) that $(B, \cdot, +, \bar{}, 0, 1)$ is a distributive lattice. This makes a Boolean algebra a complemented, distributive lattice.

2.5 Kleene algebra with tests

A Kleene algebra with tests is a Kleene algebra with an embedded Boolean algebra [1]. A Kleene algebra with tests is a tuple $(K, B, \cdot, +, ^*, \bar{}, 0, 1)$ where:

- $(K, \cdot, +, ^* 0, 1)$ is a Kleene algebra;
- $(B, \cdot, +, \bar{}, 0, 1)$ is a Boolean algebra;
- $B \subseteq K$;
- $\bar{}$ is a unary operator defined only on B.

Note that $+$, $\cdot$, 0 and 1 serve a double purpose. In the Kleene algebra they represent choice, concatenation, fail and skip, respectively. In the Boolean algebra on the other hand, they represent disjunction, conjunction, falsehood and truth, respectively. Also note that the $\cdot$ is often omitted in expressions.

Let $\Sigma = \{p_1, p_2, \ldots, p_n\}$ be the set of primitive action symbols and let $T = \{t_1, t_2, \ldots, t_m\}$ be the set of primitive test symbols. The set of Boolean expressions $BExp$ equals the set of elements of a Boolean algebra with generator T. Thus, any $b \in BExp$ can be generated by:

$$b \in BExp := 0 \mid 1 \mid t \in T \mid \bar{b} \mid b_1 + b_2 \mid b_1 \cdot b_2$$

Let Exp be the set of KAT expressions. Then, any $e \in Exp$ can be generated by:

$$e \in Exp := p \in \Sigma \mid b \in BExp \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid e^*$$

Let At be the set of atoms of T, which are all truth assignments of T. Since each element of T can either be true or false, $At = 2^T$. The number of logically distinct Boolean expressions is $2^{At} = 2^{2^T}$, since for each atom each $BExp$ can either be true or false.
We can define a partial order $\leq$ that is similar to the partial order defined on KA in Section 2.3. Let $\alpha \in At$ and $b \in BExp$, then:

$$\alpha \leq b \Leftrightarrow \alpha + b = b$$

Let GS be the set of guarded strings of the form $(At \times \Sigma)^* \times At$. Then, a guarded string is of the form $\alpha_1 p_1 \alpha_2 p_2 \ldots p_{n-1} \alpha_n$ with $\alpha_i \in At$ and $p_i \in T$. The language $L(e)$ of a KAT expression e can be described as the subset of GS that satisfies the expression [5].

Definition 2.1. *For each $e \in Exp$, the language $L(e) \subseteq GS$ of e is inductively defined as*

$$\begin{aligned} L(p) &= \{\alpha_1 p \alpha_2 \mid \alpha_1, \alpha_2 \in At\} \\ L(b) &= \{\alpha \mid \alpha \in At \wedge \alpha \leq b\} \\ L(e_1 + e_2) &= L(e_1) \cup L(e_2) \\ L(e_1 \cdot e_2) &= L(e_1) \star L(e_2) \\ L(e_1^*) &= \bigcup_{n \geq 0} L(e_1)^n \end{aligned}$$

where $F \star G$ is defined as $\{w\alpha v \mid \alpha \in At \wedge w\alpha \in F \wedge \alpha v \in G\}$, $X^0 = At$, and $X^{n+1} = X^n \star X$ for $n \geq 0$ and any set X.

It can be shown that this definition of $L(e)$ satisfies all KAT axioms mentioned in this section. For use in the following sections we give two more definitions. Firstly, we define the language $L(E)$ of a set of expressions $E \subseteq Exp$. Secondly, we define the length $||e||$ of an expression $e \in Exp$.

Definition 2.2. *For each $E \subseteq Exp$, the language $L(E) \subseteq GS$ of E is defined as*

$$L(E) = \bigcup_{e \in E} L(e)$$

Definition 2.3. *For each $e \in Exp$, the number of program symbols $||e||$ in e is inductively defined as*

$$\begin{aligned} ||b|| &= 0 \\ ||p|| &= 1 \\ ||e_1 + e_2|| &= ||e_1|| + ||e_2|| \\ ||e_1 \cdot e_2|| &= ||e_1|| + ||e_2|| \\ ||e_1^*|| &= ||e_1|| \end{aligned}$$

3 Finite automata for KAT

3.1 Design choices

In 2008 Dexter Kozen introduced two types of finite state automata on guarded strings, a DFA and an NFA [6]. A DFA on guarded strings over Σ and T is a structure

$$M_D = (Q, \Delta, \varepsilon, q_0) \ ,$$

where Q is a set of states, $q_0 \in Q$ is the initial state, and

$$\Delta : Q \to Q^{At \times \Sigma} \qquad \varepsilon : Q \to 2^{At} \ .$$

Δ is the transition function. Since the automaton is deterministic, each state has exactly one transition to another state for each input $At \times \Sigma$. Recall that a guarded string is of the form $(At \times \Sigma)^* \times At$. The $(At \times \Sigma)^*$ component then serves as input to the automaton. The At component at the end is considered the output when there is no program symbol to be read. ε is this immediate output function.

The size of the DFA explodes with increasing sizes of Σ and especially T, since $At = 2^T$. Each state has $|2^T| \cdot |\Sigma|$ outgoing transitions. For real life programs it is likely that many of these transitions will go to a trap state, and hence be more or less redundant. On top of that, recognizing the language of a DFA is often not intuitive, making it hard to work with. Therefore, we deem a DFA unsuited for our purposes.

Kozen also introduced an NFA. Based on the difference between DFAs and NFAs for KA [7, 8], one would expect a KAT NFA to have transitions of the form

$$\Delta : Q \to \mathcal{P}(Q)^{At \times \Sigma} \ .$$

Kozen's NFA is quite different, though. It has transitions of the form

$$\Delta : Q \to \mathcal{P}(Q)^{\Sigma + At} \ .$$

This means that there are two separate kinds of transitions, one for Σ and one for At. A single transition $q_i \xrightarrow{(\alpha,p)} q_j$ in the DFA corresponds to two transitions in the NFA: $q_i \xrightarrow{\alpha} q_k$ followed by $q_k \xrightarrow{p} q_j$. An advantage of this is that there is no more need for an output function. Instead, the output can be modeled with transitions of the At type. States are then accepting or not, as they are in KA automata. Due to the two kinds of transitions, this NFA is more similar to a KA NFA-Λ than it is to a KA NFA. Unfortunately, as is the case for KA NFA-Λ these automata are inconvenient to work with. Determining the language of an NFA-Λ would require a construction similar to the Λ-closure for KA. Therefore, in Subsection 3.2 we propose an NFA more similar to a KA NFA. We work with transitions similar to the $\Delta : Q \to \mathcal{P}(Q)^{At \times \Sigma}$ mentioned above, which are more similar to KA NFA transitions.

Yet, there is another design aspect where we deviate from Kozen's proposed automata, because in real life programs conditional statements rarely depend on all available tests. Let there be a branch that depends on n tests. For each truth assignment to these n tests, $2^{|T|-n}$ transitions are needed to model this branch. It is clear that this number increases exponentially for increasing $|T|$. This obfuscates the language of the automaton. On top of that, each transition requires a number of computation steps in the NFA construction. Since we avoid redundant computations, we choose to work with $BExp \times \Sigma$ rather than $At \times \Sigma$ on the transitions. This allows for summarizing these $2^{|T|-n}$ transitions into only one transition. Thus, reducing the computation steps and increasing readability.

3.2 Construction

We propose an NFA on guarded strings over Σ and T that is a labeled directed graph of the form

$$M = (Q, \Delta, \mathtt{out}, q_0),$$

where Q is a set of states, $q_0 \in Q$ is the initial state, and

$$\Delta : Q \to \mathcal{P}(Q)^{BExp \times \Sigma} \qquad \mathtt{out} : Q \to BExp \ .$$

Δ is the transition function. Each transition leads from one state to any number of states while reading an element of Σ, under condition of a $BExp$. $\mathtt{out}$ is the immediate output function. Each state has a $BExp$ as immediate output. The language $L(M)$ of such an automaton M should be read as follows.

Definition 3.1. *The language $L(M)$ of an NFA M is defined as $L(M)(q_0)$, where for every $q \in Q$*

$$\begin{aligned} \alpha \in L(M)(q) &\iff \alpha \leq \mathtt{out}(q) \\ \alpha p w \in L(M)(q) &\iff \exists \alpha \leq b : (\exists q \xrightarrow{b,p} q' \wedge w \in L(M)(q')) \end{aligned}$$

The NFA for a KAT expression e is constructed using partial derivatives. The partial derivatives for KAT are similar to the partial derivatives for KA [7]. In the NFA construction we use the following three definitions. Succeeding each definition is a lemma that shows the correctness of the definition. It is important to note that in this construction method states are labeled with expressions. Therefore, Q and Exp are often used interchangeably.

Definition 3.2. *For each $e \in Exp$, the immediate output $\mathtt{out}(e)$ is inductively defined as*

$$\begin{aligned} \mathtt{out}(b) &= b \\ \mathtt{out}(p) &= 0 \\ \mathtt{out}(e_1 + e_2) &= \mathtt{out}(e_1) + \mathtt{out}(e_2) \\ \mathtt{out}(e_1 \cdot e_2) &= \mathtt{out}(e_1) \cdot \mathtt{out}(e_2) \\ \mathtt{out}(e_1^*) &= 1 \end{aligned}$$

Lemma 3.3. *For each $e \in Exp$, $\alpha \in L(e)$ if and only if $\alpha \leq \mathtt{out}(e)$.*

Proof. By induction on the structure of e:

Base case $e = b$:

$$\begin{aligned} \alpha \in L(b) &\iff \alpha \leq b && \text{(Def. 2.1)} \\ &\iff \alpha \leq \mathtt{out}(b) && \text{(Def. 3.2)} \end{aligned}$$

Thus, the lemma holds for $e = b$.

Base case $e = p$:

$$\alpha \notin L(p) \iff \alpha \not\leq \mathtt{out}(p) = 0 \quad \text{(Def. 3.2)}$$

Thus, the lemma holds for $e = p$.

Now assume that the lemma holds for e_1 and e_2 (induction hypothesis), then:

Induction step $e = e_1 + e_2$:

$$\begin{aligned} \alpha \in L(e_1 + e_2) &\iff \alpha \in L(e_1) \cup L(e_2) && \text{(Def. 2.1)} \\ &\iff \alpha \in L(e_1) \vee \alpha \in L(e_2) \\ &\iff \alpha \leq \mathtt{out}(e_1) \vee \alpha \leq \mathtt{out}(e_2) && \text{(IH)} \\ &\iff \alpha \leq \mathtt{out}(e_1) + \mathtt{out}(e_2) && (\alpha \in At) \\ &\iff \alpha \leq \mathtt{out}(e_1 + e_2) && \text{(Def. 3.2)} \end{aligned}$$

Thus, the lemma holds for $e = e_1 + e_2$.

Induction step $e = e_1 \cdot e_2$:

$$\begin{aligned} \alpha \in L(e_1 \cdot e_2) &\iff \alpha \in L(e_1) \star L(e_2) && \text{(Def. 2.1)} \\ &\iff \alpha \in L(e_1) \wedge \alpha \in L(e_2) \\ &\iff \alpha \leq \mathtt{out}(e_1) \wedge \alpha \leq \mathtt{out}(e_2) && \text{(IH)} \\ &\iff \alpha \leq \mathtt{out}(e_1) \cdot \mathtt{out}(e_2) \\ &\iff \alpha \leq \mathtt{out}(e_1 \cdot e_2) && \text{(Def. 3.2)} \end{aligned}$$

Thus, the lemma holds for $e = e_1 \cdot e_2$.

Induction step $e = e_1^*$:

$$\alpha \in L(e_1^*) \iff \alpha \leq \mathtt{out}(e_1^*) = 1 \text{ (Def. 3.2)}$$

Thus, the lemma holds for $e = e_1^*$.

□

Definition 3.4. *For each $e \in Exp$, $b \in BExp$ and $p \in \Sigma$, the partial derivatives $\delta_{bp}(e)$ of e are inductively defined as follows, with $\mathtt{out}(e)$ as in Def. 3.2.*

$$\begin{aligned} \delta_{bp}(b) &= \emptyset \\ \delta_{bp}(q) &= \begin{cases} \{1\} & \text{if } p = q \\ \emptyset & \text{otherwise} \end{cases} \\ \delta_{bp}(e_1 + e_2) &= \delta_{bp}(e_1) \cup \delta_{bp}(e_2) \\ \delta_{bp}(e_1 \cdot e_2) &= \begin{cases} \delta_{bp}(e_1) \cdot e_2 \cup \delta_{bp}(e_2) & \text{if } \mathtt{out}(e_1) \neq 0 \\ \delta_{bp}(e_1) \cdot e_2 & \text{otherwise} \end{cases} \\ \delta_{bp}(e^*) &= \delta_{bp}(e) \cdot e^* \end{aligned}$$

Lemma 3.5. *For each $e \in Exp$, $\exists \alpha \leq b : \alpha p w \in L(e)$ if and only if $w \in L(\delta_{bp}(e))$.*

Proof. By induction on the structure of e:

Base case $e = b$:

$$\begin{aligned} \neg\exists \alpha \leq b : \alpha p w \in L(b) &\iff \forall \alpha \leq b : \alpha p w \notin L(b) \\ &\iff w \notin L(\delta_{bp}(b)) = L(\emptyset) && \text{(Def. 3.4)} \end{aligned}$$

Thus, the lemma holds for $e = b$.

Base case $e = q$:

$$\begin{aligned} \exists \alpha \leq b : \alpha p w \in L(q) &\iff \exists \alpha \leq b : \alpha p w \in L(q) \wedge w = \beta \wedge p = q && \text{(Def. 2.1)} \\ &\iff \delta_{bp}(q) = \{1\} \wedge w = \beta && \text{(Def. 3.4)} \\ &\iff w \in L(\delta_{bp}(q)) && \text{(Def. 3.4)} \end{aligned}$$

Thus, the lemma holds for $e = q$.

Now assume that the lemma holds for e_1 and e_2 (induction hypothesis), then:

Induction step $e = e_1 + e_2$:

$$\begin{aligned}
\exists \alpha \leq b : \alpha p w \in L(e_1 + e_2) &\iff \exists \alpha \leq b : \alpha p w \in L(e_1) \cup L(e_2) && \text{(Def. 2.1)} \\
&\iff \exists \alpha \leq b : (\alpha p w \in L(e_1) \vee \alpha p w \in L(e_2)) \\
&\iff \exists \alpha \leq b : \alpha p w \in L(e_1) \vee \exists \alpha \leq b : \alpha p w \in L(e_2) \\
&\iff w \in L(\delta_{bp}(e_1)) \vee w \in L(\delta_{bp}(e_2)) && \text{(IH)} \\
&\iff w \in L(\delta_{bp}(e_1)) \cup L(\delta_{bp}(e_2)) \\
&\iff w \in L(\delta_{bp}(e_1) \cup \delta_{bp}(e_2)) && \text{(Def. 2.2)} \\
&\iff w \in L(\delta_{bp}(e_1 + e_2)) && \text{(Def. 3.4)}
\end{aligned}$$

Thus, the lemma holds for $e = e_1 + e_2$.

Induction step $e = e_1 \cdot e_2$: Consider $\alpha p w \in L(e_1 \cdot e_2) = L(e_1) \star L(e_2)$. This means that $\alpha p w = u \beta v$ such that $u\beta \in L(e_1)$ and $\beta v \in L(e_2)$ (Def. 2.1). We proceed by case analysis:

case (1)	$\|\|u\|\| = 0$	This means that $\alpha = \beta$ and thus $\alpha \in L(e_1)$ and $\alpha p w \in L(e_2)$.
case (2)	$\|\|u\|\| \neq 0$	This means that $u = \alpha p u'$ and $\alpha p u' \beta \in L(e_1)$ and $\beta v \in L(e_2)$ with $u' \beta v = w$.

For case (1) we have:

$$\begin{aligned}
\exists \alpha \leq b : \alpha p w \in L(e_1 \cdot e_2) &\iff \exists \alpha \leq b : \alpha p w \in L(e_1) \star L(e_2) && \text{(Def. 2.1)} \\
&\iff \exists \alpha \leq b : (\alpha \in L(e_1) \wedge \alpha p w \in L(e_2)) && \text{(case (1))} \\
&\iff \exists \alpha \leq b : (\alpha \in L(e_1) \wedge w \in L(\delta_{bp}(e_2))) && \text{(IH)} \\
&\iff \exists \alpha \leq b : \alpha \in L(e_1) \wedge w \in L(\delta_{bp}(e_2)) \\
&\iff \exists \alpha \leq b : \alpha \leq \mathtt{out}(e_1) \wedge w \in L(\delta_{bp}(e_2)) && \text{(Lemma 3.5)} \\
&\iff \mathtt{out}(e_1) \neq 0 \wedge w \in L(\delta_{bp}(e_2))
\end{aligned}$$

For case (2) we have:

$$\begin{aligned}
\exists \alpha \leq b : \alpha p w \in L(e_1 \cdot e_2) &\iff \exists \alpha \leq b : \alpha p w \in L(e_1) \star L(e_2) && \text{(Def. 2.1)} \\
&\iff \exists \alpha \leq b : (\alpha p u' \beta \in L(e_1) \wedge \beta v \in L(e_2)) && \text{(case (2))} \\
&\iff \exists \alpha \leq b : \alpha p u' \beta \in L(e_1) \wedge \beta v \in L(e_2) \\
&\iff u' \beta \in L(\delta_{bp}(e_1)) \wedge \beta v \in L(e_2) && \text{(IH)} \\
&\iff u' \beta v \in L(\delta_{bp}(e_1)) \star L(e_2) && \text{(Def. 2.1)} \\
&\iff u' \beta v \in L(\delta_{bp}(e_1) \cdot e_2) && \text{(Def. 2.1)} \\
&\iff w \in L(\delta_{bp}(e_1) \cdot e_2) && \text{(case (2))}
\end{aligned}$$

If $\mathtt{out}(e_1) \neq 0$, both case (1) and case (2) may apply. Then, $\exists \alpha \leq b : \alpha p w \in L(e_1 \cdot e_2)$ if and only if $w \in L(\delta_{bp}(e_1) \cdot e_2 \cup \delta_{bp}(e_1))$. Otherwise (when $\mathtt{out}(e_1) = 0$), only case (2) can apply. Then, $\exists \alpha \leq b : \alpha p w \in L(e_1 \cdot e_2)$ if and only if $w \in L(\delta_{bp}(e_1) \cdot e_2)$. This coincides with the cases in the definition. Thus, the lemma holds for $e = e_1 \cdot e_2$.

Induction step $e = e_1^*$: Consider $\alpha p w \in L(e_1^*) = \bigcup_{n \geq 0} L(e_1)^n$. Thus, $\alpha p w \in L(e_1)^k$ for some $k \geq 0$. It cannot be $k = 0$ since $\alpha p w \notin L(e_1)^0 = At$. Hence, $\alpha p w \in L(e_1) \star L(e_1)^{k-1}$ for some $k > 0$. This means that $\alpha p v \beta \in L(e_1)$ such that $\alpha p w = \alpha p v \beta u \in L(e_1)^k$ with $w = v \beta u$ (Def. 2.1). Then:

$$\begin{aligned}
\exists \alpha \leq b : \alpha p w \in L(e_1^*) &\iff \exists \alpha \leq b : \alpha p v \beta u \in L(e_1^*) && (w = v \beta u) \\
&\iff \exists \alpha \leq b : \alpha p v \beta u \in L(e_1) \star L(e_1^*) && \text{(Def. 2.1, } k > 0) \\
&\iff \exists \alpha \leq b : (\alpha p v \beta \in L(e_1) \wedge \beta u \in L(e_1^*)) && \text{(Def. 2.1)} \\
&\iff \exists \alpha \leq b : \alpha p v \beta \in L(e_1) \wedge \beta u \in L(e_1^*) \\
&\iff v \beta \in L(\delta_{bp}(e_1)) \wedge \beta u \in L(e_1^*) && \text{(IH)} \\
&\iff v \beta u \in L(\delta_{bp}(e_1)) \star L(e_1^*) && \text{(Def. 2.1)} \\
&\iff v \beta u \in L(\delta_{bp}(e_1) \cdot e_1^*) && \text{(Def. 2.1)} \\
&\iff w \in L(\delta_{bp}(e_1) \cdot e_1^*) && (w = v \beta u)
\end{aligned}$$

It follows that $\exists \alpha \leq b : \alpha p w \in L(e_1^*)$ if and only if $w \in L(\delta_{bp}(e_1) \cdot e_1^*) = L(\delta_{bp}(e_1^*))$ (Def. 3.4). Thus, the lemma holds for $e = e_1^*$.

□

This definition of $BExp \times \Sigma$ type derivatives has a drawback. It does not take into account contradictions.

A simple example shows this:

$$\begin{aligned}\delta_{t_1 p}(\bar{t_1} p) &= \delta_{t_1 p}(\bar{t_1}) \cdot p \cup \delta_{t_1 p}(p) \\ &= \emptyset \cdot p \cup \{1\} \\ &= \{1\}\end{aligned}$$

It is clear that t_1 and $\bar{t_1}$ form a contradiction. Therefore, there should be no derivative $\delta_{t_1 p}(\bar{t_1} p)$. If we were to prevent this, we would need to check for each $\delta_{b_1 p}(b_2 q w)$ whether b_1 satisfies b_2. Unfortunately, the satisfiability problem is NP-hard. Since we attempt to find an efficient algorithm for constructing an NFA for KAT, we accept this inaccuracy in the derivatives. In our construction we work with the `first` set of an expression, see Def. 3.9. This means that the above situation nevers occurs in our construction, because contradictory expressions are not in the `first` set of an expression. E.g. $(t_1, p) \notin \texttt{first}(\bar{t_1} p)$. It has to be noted, though, the these derivatives are not suited for generic use.

Definition 3.6. *For each $w \in (BExp \times \Sigma)^*$, the word derivatives $\delta_w(e)$ of e are defined as*

$$\begin{aligned}\delta_1(e) &= \{e\} \\ \delta_{bpw}(e) &= \bigcup_{e' \in \delta_{bp}(e)} \delta_w(e')\end{aligned}$$

Lemma 3.7. *For each $w \in (BExp \times \Sigma)^+$, $e \in Exp$, $\delta_w(e)$ behaves as follows.*

$$\begin{aligned}\delta_w(b) &= \emptyset \\ \delta_w(q) &= \begin{cases} \{1\} & \text{if } p = q \wedge w = bp \\ \emptyset & \text{otherwise} \end{cases} \\ \delta_w(e_1 + e_2) &= \delta_w(e_1) \cup \delta_w(e_2) \\ \delta_w(e_1 \cdot e_2) &\subseteq \delta_w(e_1) \cdot e_2 \cup \bigcup_{w = uv \wedge v \neq 1} \delta_v(e_2) \\ \delta_w(e^*) &\subseteq \bigcup_{w = uv \wedge v \neq 1} \delta_v(e) \cdot e^*\end{aligned}$$

Proof. We proceed by case analysis.

Case $e = b$:

$$\begin{aligned}\delta_w(b) &= \textstyle\bigcup_{e' \in \delta_{bp}(b)} \delta_v(e') && \text{(Def. 3.6, } w \in (BExp \times \Sigma)^+\text{)} \\ &= \emptyset && \text{(Def. 3.4)}\end{aligned}$$

Thus, the lemma holds for $e = b$.

Case $e = q$:

$$\begin{aligned}\delta_w(q) &= \textstyle\bigcup_{e' \in \delta_{bp}(q)} \delta_v(e') && \text{(Def. 3.6, } w \in (BExp \times \Sigma)^+\text{)} \\ &= \begin{cases} \delta_v(\{1\}) & \text{if } p = q \\ \emptyset & \text{otherwise} \end{cases} && \text{(Def. 3.4)} \\ &= \begin{cases} \{1\} & \text{if } p = q \wedge w = bp \\ \emptyset & \text{otherwise} \end{cases} && \text{(Def. 3.6)}\end{aligned}$$

Thus, the lemma holds for $e = q$.

Case $e = e_1 + e_2$:

$$\begin{aligned}\delta_w(e_1 + e_2) &= \textstyle\bigcup_{e' \in \delta_{bp}(e_1 + e_2)} \delta_v(e') && \text{(Def. 3.6, } w \in (BExp \times \Sigma)^+\text{)} \\ &= \textstyle\bigcup_{e' \in \delta_{bp}(e_1)} \delta_v(e') \cup \bigcup_{e' \in \delta_{bp}(e_2)} \delta_v(e') && \text{(Def. 3.4)} \\ &= \delta_w(e_1) \cup \delta_w(e_2) && \text{(Def. 3.6, } w = bpv\text{)}\end{aligned}$$

Thus, the lemma holds for $e = e_1 + e_2$.

Case $e = e_1 \cdot e_2$: We proceed by induction on the length of w. Since $w \in (BExp \times \Sigma)^+$, $||w|| = 1$ and thus $w = bp$ is the base case.

$$\begin{aligned}\delta_{bp}(e_1 \cdot e_2) &\subseteq \delta_{bp}(e_1) \cdot e_2 \cup \delta_{bp}(e_2) && \text{(Def. 3.4)} \\ &= \delta_w(e_1) \cdot e_2 \cup \delta_w(e_2) && (w = bp) \\ &\subseteq \delta_w(e_1) \cdot e_2 \cup \textstyle\bigcup_{w = uv \wedge v \neq 1} \delta_v(e_2)\end{aligned}$$

Now assume that the lemma holds for all w' such that $||w'|| < ||w||$ (induction hypothesis) and take $w = bpv$, then:

$$\begin{aligned}
\delta_w(e_1 \cdot e_2) &= \textstyle\bigcup_{e' \in \delta_{bp}(e_1 \cdot e_2)} \delta_v(e') && \text{(Def. 3.6)}\\
&\subseteq \textstyle\bigcup_{e' \in \delta_{bp}(e_1) \cdot e_2} \delta_v(e') \cup \bigcup_{e' \in \delta_{bp}(e_2)} \delta_v(e') && \text{(Def. 3.4)}\\
&= \textstyle\bigcup_{e' \in \delta_{bp}(e_1)} \delta_v(e' \cdot e_2) \cup \bigcup_{e' \in \delta_{bp}(e_2)} \delta_v(e')\\
&\subseteq \textstyle\bigcup_{e' \in \delta_{bp}(e_1)} \left(\delta_v(e') \cdot e_2 \cup \bigcup_{v = uv' \wedge v' \neq 1} \delta_{v'}(e_2)\right) \cup \bigcup_{e' \in \delta_{bp}(e_2)} \delta_v(e') && \text{(IH)}\\
&= \textstyle\bigcup_{e' \in \delta_{bp}(e_1)} \delta_v(e') \cdot e_2 \cup \bigcup_{v = uv' \wedge v' \neq 1} \delta_{v'}(e_2) \cup \bigcup_{e' \in \delta_{bp}(e_2)} \delta_v(e')\\
&= \delta_w(e_1) \cdot e_2 \cup \textstyle\bigcup_{v = uv' \wedge v' \neq 1} \delta_{v'}(e_2) \cup \delta_w(e_2) && \text{(Def. 3.6)}\\
&\subseteq \delta_w(e_1) \cdot e_2 \cup \textstyle\bigcup_{w = uv \wedge v \neq 1} \delta_v(e_2)
\end{aligned}$$

Thus, the lemma holds for $e = e_1 \cdot e_2$.

Case $e = e^*$: We proceed by induction on the length of w. Since $w \in (BExp \times \Sigma)^+$, $||w|| = 1$ and thus $w = bp$ is the base case.

$$\begin{aligned}
\delta_{bp}(e^*) &= \delta_{bp}(e) \cdot e^* && \text{(Def. 3.4)}\\
&= \delta_w(e) \cdot e^* && (w = bp)\\
&\subseteq \textstyle\bigcup_{w = uv \wedge v \neq 1} \delta_v(e) \cdot e^*
\end{aligned}$$

Now assume that the lemma holds for all w' such that $||w'|| < ||w||$ (induction hypothesis) and take $w = bpv$, then:

$$\begin{aligned}
\delta_w(e^*) &= \textstyle\bigcup_{e' \in \delta_{bp}(e^*)} \delta_v(e') && \text{(Def. 3.6)}\\
&= \textstyle\bigcup_{e' \in \delta_{bp}(e) \cdot e^*} \delta_v(e') && \text{(Def. 3.4)}\\
&= \textstyle\bigcup_{e' \in \delta_{bp}(e)} \delta_v(e' \cdot e^*)\\
&\subseteq \textstyle\bigcup_{e' \in \delta_{bp}(e)} \left(\delta_v(e') \cdot e^* \cup \bigcup_{v = uv' \wedge v' \neq 1} \delta_{v'}(e^*)\right) && \text{(case } e = e_1 \cdot e_2\text{)}\\
&= \textstyle\bigcup_{e' \in \delta_{bp}(e)} \delta_v(e') \cdot e^* \cup \bigcup_{v = uv' \wedge v' \neq 1} \delta_{v'}(e^*)\\
&= \delta_w(e) \cdot e^* \cup \textstyle\bigcup_{v = uv' \wedge v' \neq 1} \delta_{v'}(e^*) && \text{(Def. 3.6)}\\
&\subseteq \delta_w(e) \cdot e^* \cup \textstyle\bigcup_{v = uv' \wedge v' \neq 1} \bigcup_{v' = u'w' \wedge w' \neq 1} \delta_{w'}(e) \cdot e^* && \text{(IH)}\\
&= \delta_w(e) \cdot e^* \cup \textstyle\bigcup_{v = uw' \wedge w' \neq 1} \delta_{w'}(e) \cdot e^*\\
&\subseteq \textstyle\bigcup_{w = uv \wedge v \neq 1} \delta_v(e) \cdot e^*
\end{aligned}$$

Thus, the lemma holds for $e = e^*$.

□

Theorem 3.8. *For each $e \in Exp$ and $w \in (At \times \Sigma)^*$, $w\alpha \in L(e)$ if and only if $w\alpha \in L(M)(e)$.*

Proof. We proceed by induction on the length of w.

Base case If $||w|| = 0$, then:

$$\begin{aligned}
w\alpha \in L(e) &\iff \alpha \in L(e) && (||w|| = 0)\\
&\iff \alpha \leq \mathtt{out}(e) && \text{(Lemma 3.3)}\\
&\iff \alpha \in L(M)(e) && \text{(Def. 3.1)}\\
&\iff w\alpha \in L(M)(e) && (||w|| = 0)
\end{aligned}$$

Induction step Now assume that the lemma holds for all w' such that $||w'|| < ||w||$ (induction hypothesis) and take $w = bpv$, then:

$$\begin{aligned}
w\alpha \in L(e) &\iff \beta pv \in L(e) \wedge w\alpha = \beta pv && (w = \beta pv)\\
&\iff \exists \beta \leq b : v \in L(\delta_{bp}(e)) && \text{(Lemma 3.5)}\\
&\iff \exists \beta \leq b : v \in L(M)(\delta_{bp}(e)) && \text{(IH)}\\
&\iff \exists \beta \leq b : (\exists e' \in \delta_{bp}(e) : v \in L(M)(e')) && \text{(Def. 2.2)}\\
&\iff \exists \beta \leq b : (\exists e \xrightarrow{b,p} e' \wedge v \in L(M)(e')) && \text{(construction)}\\
&\iff \beta pv \in L(M)(e) && \text{(Def. 3.1)}\\
&\iff w\alpha \in L(M)(e) && (w = \beta pv)
\end{aligned}$$

□

Unlike previous works that have partial derivatives of the form $\delta_{\alpha p}(e)$ [6], we work with $\delta_{bp}(e)$. Since $BExp = 2^{At}$, this would increase the number of derivatives to be computed by an order of magnitude. The number of computations can be limited by determining the $\texttt{first}$ set of an expression [9]. $\texttt{first}(e)$ contains all input tuples $BExp \times \Sigma$ that could be read at that point in time. The derivatives of e only have to be computed for the input tuples in $\texttt{first}(e)$, not for all $BExp \times \Sigma$. If some $(b, p) \notin \texttt{first}(e)$ and b is not contained in another b_1 such that $(b_1, p) \in \texttt{first}(e)$, then $\delta_{bp}(e) = \emptyset$. Using this short-circuiting dramatically decreases the number of computation steps needed in the NFA construction.

Definition 3.9. *For each $e \in Exp$, the $\texttt{first}(e)$ of e is inductively defined as*

$$\begin{aligned}
\texttt{first}(b) &= \emptyset \\
\texttt{first}(p) &= \{(1, p)\} \\
\texttt{first}(e_1 + e_2) &= \texttt{first}(e_1) \cup \texttt{first}(e_2) \\
\texttt{first}(e_1 \cdot e_2) &= \texttt{first}(e_1) \cup \texttt{out}(e_1) \cdot_1 \texttt{first}(e_2) \\
\texttt{first}(e^*) &= \texttt{first}(e)
\end{aligned}$$

where $b \cdot_1 X = \{(bb', p) \mid (b', p) \in X\}$.

Lemma 3.10. *For each $e \in Exp$, $\alpha p w \in L(e)$ if and only if $\exists (b, p) \in \texttt{first}(e) : \alpha \leq b$.*

Proof. By induction on the structure of e:

Base case $e = b$:

$$\begin{aligned}
\alpha p w \notin L(b) &\iff \texttt{first}(b) = \emptyset && \text{(Def. 3.9)} \\
&\iff \neg \exists (b, p) \in \texttt{first}(b) : \alpha \leq b
\end{aligned}$$

Thus, the lemma holds for $e = b$.

Base case $e = p$:

$$\begin{aligned}
\alpha p w \in L(p) &\iff \texttt{first}(p) = \{(1, p)\} && \text{(Def. 3.9)} \\
&\iff \exists (b, p) \in \texttt{first}(p) : \alpha \leq b
\end{aligned}$$

Thus, the lemma holds for $e = p$.

Now assume that the lemma holds for e_1 and e_2 (induction hypothesis), then:

Induction step $e = e_1 + e_2$:

$$\begin{aligned}
\alpha p w \in L(e_1 + e_2) &\iff \alpha p w \in L(e_1) \cup L(e_2) && \text{(Def. 2.1)} \\
&\iff \alpha p w \in L(e_1) \vee \alpha p w \in L(e_1) \\
&\iff \exists (b, p) \in \texttt{first}(e_1) : \alpha \leq b \vee \exists (b, p) \in \texttt{first}(e_2) : \alpha \leq b && \text{(IH)} \\
&\iff \exists (b, p) \in \texttt{first}(e_1) \cup \texttt{first}(e_2) : \alpha \leq b \\
&\iff \exists (b, p) \in \texttt{first}(e_1 + e_2) : \alpha \leq b && \text{(Def. 3.9)}
\end{aligned}$$

Thus, the lemma holds for $e = e_1 + e_2$.

Induction step $e = e_1 \cdot e_2$: Consider $\alpha p w \in L(e_1 \cdot e_2) = L(e_1) \star L(e_2)$. This means that $\alpha p w = u \beta v$ such that $u\beta \in L(e_1)$ and $\beta v \in L(e_2)$ (Def. 2.1). We proceed by case analysis:

case (1) $||u|| = 0$ This means that $\alpha = \beta$ and thus $\alpha \in L(e_1)$ and $\alpha p w \in L(e_2)$.
case (2) $||u|| \neq 0$ This means that $u = \alpha p u'$ and $\alpha p u' \beta \in L(e_1)$ and $\beta v \in L(e_2)$ with $u' \beta v = w$.

For case (1) we have:

$$\begin{aligned}
\alpha p w \in L(e_1 \cdot e_2) &\iff \alpha \in L(e_1) \wedge \alpha p w \in L(e_2) && \text{(case (1))} \\
&\iff \alpha \leq \texttt{out}(e_1) \wedge \alpha p w \in L(e_2) && \text{(Lemma 3.3)} \\
&\iff \alpha \leq \texttt{out}(e_1) \wedge \exists (b, p) \in \texttt{first}(e_2) : \alpha \leq b && \text{(IH)} \\
&\iff \exists (b, p) \in \texttt{out}(e_1) \cdot_1 \texttt{first}(e_2) : \alpha \leq b && \text{(Def. 3.9)}
\end{aligned}$$

For case (2) we have:

$$\begin{aligned}
\alpha p w \in L(e_1 \cdot e_2) &\iff \alpha p u' \beta \in L(e_1) \wedge \beta v \in L(e_2) && \text{(case (2))} \\
&\iff \exists (b, p) \in \texttt{first}(e_1) : \alpha \leq b \wedge \beta v \in L(e_2) && \text{(IH)}
\end{aligned}$$

It follows that $\alpha p w \in L(e_1 \cdot e_2)$ if and only if $\exists (b, p) \in \texttt{first}(e_1) \cup \texttt{out}(e_1) \cdot_1 \texttt{first}(e_2) : \alpha \leq b$ and hence by definition $\exists (b, p) \in \texttt{first}(e_1 \cdot e_2) : \alpha \leq b$. Thus, the lemma holds for $e = e_1 \cdot e_2$.

Induction step $e = e_1^*$: Consider $\alpha pw \in L(e_1^*) = \bigcup_{n \geq 0} L(e_1)^n$. Thus, $\alpha pw \in L(e_1)^k$ for some $k \geq 0$. It cannot be $k = 0$ since $\alpha pw \notin L(e_1)^0 = At$. Hence, $\alpha pw \in L(e_1) \star L(e_1)^{k-1}$ for some $k > 0$. This means that $\alpha pv\beta \in L(e_1)$ such that $\alpha pw = \alpha pv\beta u \in L(e_1^*)$ with $w = v\beta u$. (Def. 2.1) Then:

$$\begin{array}{lll} \alpha pw \in L(e_1^*) & \iff \alpha pv\beta u \in L(e_1^*) & (w = v\beta u) \\ & \iff \alpha pv\beta \in L(e_1) & (\text{Def. 2.1}) \\ & \iff \exists (b,p) \in \texttt{first}(e_1) : \alpha \leq b & (\text{IH}) \\ & \iff \exists (b,p) \in \texttt{first}(e_1^*) : \alpha \leq b & (\text{Def. 3.9}) \end{array}$$

Thus, the lemma holds for $e = e_1^*$.

□

Now we can construct the whole NFA $M = (Q, \Delta, o, q_0)$ from a given KAT expression e:

$$\begin{aligned} Q :&\ \bigcup_w \delta_w(e) \\ \Delta :&\ Q \to \mathcal{P}(Q)^{BExp \times \Sigma},\ \text{if } (b,p) \in \texttt{first}(q) \text{ then } q \mapsto \delta_{bp}(q) \\ \texttt{out} :&\ Q \to BExp,\ q \mapsto \texttt{out}(q) \\ q_0 :&\ e \end{aligned}$$

with $w \in (BExp \times \Sigma)^*$.

This is equivalent to execution of the following algorithm. The next sections build on an implementation similar to this pseudocode. Note Q in this algorithm is not precisely the set of states but rather a tuple of a state (its label) and its immediate output. After execution of the algorithm Q contains the states of the automaton, tuples of the form $Exp \times BExp$. Δ contains all transitions of the automaton. These are tuples of the form $Exp \times BExp \times \Sigma \times Exp$.

INPUT: Exp
OUTPUT: $Q : Exp \times BExp, \Delta : Exp \times BExp \times \Sigma \times Exp$

```
1  Q is empty; todo is empty; Δ is empty;
2  insert (e, out(e)) in Q;
3  insert e in todo;
4  while todo is not empty do
5      extract x from todo;
6      for all (b, p) ∈ first(x) do
7          for all d ∈ δ_bp(x) do
8              if (d, out(d)) ∉ Q then
9                  insert (d, out(d)) in Q;
10                 insert d in todo;
11             fi
12             insert (x, b, p, d) in Δ;
13         od
14     od
15 od
```

3.3 Complexity

Since the main aim of this thesis is to efficiently compute the equivalence relation between two NFAs, it is interesting to see how the size of our NFAs relates to other the size of other NFAs. Therefore, we prove the following two theorems about the number of states and transitions, respectively.

Theorem 3.11. *For each $e \in Exp$, $|\bigcup_w \delta_w(e)| \leq ||e|| + 1$ where $w \in (BExp \times \Sigma)^*$.*

Proof. Note that $|\delta_1(e)| = |\{e\}| = 1$ for all $e \in Exp$. Therefore, it suffices to prove that $|\bigcup_w \delta_w(e)| \leq ||e||$ for $w \in (BExp \times \Sigma)^+$. By induction on the structure of e:

Base case $e = b$:

$$\begin{aligned} |\bigcup_w \delta_w(b)| &= |\emptyset| && \text{(Lemma 3.7)} \\ &= 0 \\ &\leq ||b|| && \text{(Def. 2.3)} \end{aligned}$$

Thus, the theorem holds for $e = b$.

Base case $e = p$:

$$\begin{aligned} |\bigcup_w \delta_w(p)| &= |\delta_{1_p}(p)| && \text{(Lemma 3.7)} \\ &= |\{1\}| \\ &= 1 \\ &\leq ||p|| && \text{(Def. 2.3)} \end{aligned}$$

Thus, the theorem holds for $e = p$.

Now assume that the theorem holds for e_1 and e_2 (induction hypothesis), then:

Induction step $e = e_1 + e_2$:

$$\begin{aligned} |\bigcup_w \delta_w(e_1 + e_2)| &= |\bigcup_w (\delta_w(e_1) \cup \delta_w(e_2))| && \text{(Lemma 3.7)} \\ &\leq |\bigcup_w \delta_w(e_1)| + |\bigcup_w \delta_w(e_2)| \\ &\leq ||e_1|| + ||e_2|| && \text{(IH)} \\ &\leq ||e_1 + e_2|| && \text{(Def. 2.3)} \end{aligned}$$

Thus, the theorem holds for $e = e_1 + e_2$.

Induction step $e = e_1 \cdot e_2$:

$$\begin{aligned} |\bigcup_w \delta_w(e_1 \cdot e_2)| &\leq |\bigcup_w (\delta_w(e_1) \cdot e_2 \cup \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_2))| && \text{(Lemma 3.7)} \\ &\leq |\bigcup_w (\delta_w(e_1) \cdot e_2 \cup \bigcup_w \delta_w(e_2)| \\ &\leq |\bigcup_w \delta_w(e_1) \cdot e_2| + |\bigcup_w \delta_w(e_2)| \\ &= |\bigcup_w \delta_w(e_1)| + |\bigcup_w \delta_w(e_2)| \\ &\leq ||e_1|| + ||e_2|| && \text{(IH)} \\ &\leq ||e_1 \cdot e_2|| && \text{(Def. 2.3)} \end{aligned}$$

Thus, the theorem holds for $e = e_1 \cdot e_2$.

Induction step $e = e_1^*$:

$$\begin{aligned} |\bigcup_w \delta_w(e_1^*)| &\leq |\bigcup_{w=uv \wedge v \neq 1} \delta_v(e_1) \cdot e_1^*| && \text{(Lemma 3.7)} \\ &\leq |\bigcup_w \delta_v(e_1) \cdot e_1^*| \\ &= |\bigcup_w \delta_w(e_1)| \\ &\leq ||e_1|| && \text{(IH)} \\ &\leq ||e_1^*|| && \text{(Def. 2.3)} \end{aligned}$$

Thus, the theorem holds for $e = e_1^*$.

□

Theorem 3.12. *For each $e \in Exp$, $|\bigcup_w \bigcup_{e' \in \delta_w(e)} \texttt{first}(e')| \leq ||e||^2$ where $w \in (BExp \times \Sigma)^*$.*

Proof. By induction on the structure of e:

Base case $e = b$:

$$\begin{aligned} |\bigcup_w \bigcup_{e' \in \delta_w(b)} \texttt{first}(e')| &= |\emptyset| && \text{(Def. 3.9)} \\ &= 0 \\ &\leq ||b||^2 && \text{(Def. 2.3)} \end{aligned}$$

Thus, the theorem holds for $e = b$.

Base case $e = p$:

$$\begin{aligned} |\bigcup_w \bigcup_{e' \in \delta_w(p)} \texttt{first}(e')| &= |\{1, p\}| && \text{(Def. 3.9)} \\ &= 1 \\ &\leq ||p||^2 && \text{(Def. 2.3)} \end{aligned}$$

Thus, the theorem holds for $e = p$.

Now assume that the theorem holds for e_1 and e_2 (induction hypothesis), then:

Induction step $e = e_1 + e_2$:

$|\bigcup_w \bigcup_{e' \in \delta_w(e_1 + e_2)} \texttt{first}(e')|$

$= |\bigcup_w \bigcup_{e' \in \delta_w(e_1)} \mathtt{first}(e') \cup \bigcup_w \bigcup_{e' \in \delta_w(e_2)} \mathtt{first}(e')|$ (Lemma 3.7)
$\leq |\bigcup_w \bigcup_{e' \in \delta_w(e_1)} \mathtt{first}(e')| + |\bigcup_w \bigcup_{e' \in \delta_w(e_2)} \mathtt{first}(e')|$
$\leq ||e_1||^2 + ||e_2||^2$ (IH)
$\leq ||e_1 + e_2||^2$ (Def. 2.3)

Thus, the theorem holds for $e = e_1 + e_2$.

Induction step $e = e_1 \cdot e_2$:

$|\bigcup_w \bigcup_{e' \in \delta_w(e_1 \cdot e_2)} \mathtt{first}(e')|$
$\leq |\bigcup_w \bigcup_{e' \in \delta_w(e_1) \cdot e_2} \mathtt{first}(e') \cup \bigcup_w \bigcup_{e' \in \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_2)} \mathtt{first}(e')|$ (Lemma 3.7)
$\leq |\bigcup_w \bigcup_{e' \in \delta_w(e_1) \cdot e_2} \mathtt{first}(e')| + |\bigcup_w \bigcup_{e' \in \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_2)} \mathtt{first}(e')|$
$= |\bigcup_w \bigcup_{e' \in \delta_w(e_1)} \mathtt{first}(e' \cdot e_2)| + |\bigcup_w \bigcup_{e' \in \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_2)} \mathtt{first}(e')|$
$\leq |\bigcup_w \bigcup_{e' \in \delta_w(e_1)} \mathtt{first}(e')| + |\mathtt{first}(e_2)| + |\bigcup_w \bigcup_{e' \in \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_2)} \mathtt{first}(e')|$
$= |\bigcup_w \bigcup_{e' \in \delta_w(e_1)} \mathtt{first}(e')| + |\bigcup_w \bigcup_{e' \in \delta_w(e_2)} \mathtt{first}(e')|$
$\leq ||e_1||^2 + ||e_2||^2$ (IH)
$\leq ||e_1 \cdot e_2||^2$ (Def. 2.3)

Thus, the theorem holds for $e = e_1 \cdot e_2$.

Induction step $e = e_1^*$:

$|\bigcup_w \bigcup_{e' \in \delta_w(e_1^*)} \mathtt{first}(e')|$
$= |\bigcup_w \bigcup_{e' \in \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_1) \cdot e_1^*} \mathtt{first}(e')|$ (Lemma 3.7)
$= |\bigcup_w \bigcup_{e' \in \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_1)} \mathtt{first}(e' \cdot e_1^*)|$
$\leq |\bigcup_w \bigcup_{e' \in \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_1)} \mathtt{first}(e')| + |\mathtt{first}(e_1^*)|$
$\leq |\bigcup_w \bigcup_{e' \in \bigcup_{w=uv \wedge v \neq 1} \delta_v(e_1)} \mathtt{first}(e')| + |\mathtt{first}(e_1)|$ (Def. 3.9)
$\leq |\bigcup_w \bigcup_{e' \in \delta_w(e_1)} \mathtt{first}(e')|$
$\leq ||e_1||^2$ (IH)
$\leq ||e_1^*||^2$ (Def. 2.3)

Thus, the theorem holds for $e = e_1^*$.

□

Theorem 3.11 shows that an NFA constructed with the described method has at most $||e|| + 1$ states. Thus, the size of the NFA is independent of the primitive test symbols. This means that the KAT NFA construction method has the same descriptional complexity behavior as the usual KA NFA construction method [7]. From Theorem 3.12 it follows that such an NFA has at most $||e||^2$ transitions. However, these numbers are frequently lower.

4 Equivalence

The problem in equivalence checking is to determine for two automata M_1 and M_2 if $L(M_1) = L(M_2)$. A naive approach to NFA equivalence checking is to determinize M_1 and M_2 first. Typically, this is done with the powerset construction [10]. The resulting DFAs are then minimized. A minimized DFA is unique but for perhaps the naming of the states [11]. Since minimization preserves the language of an automaton, the resulting DFAs are equal if and only if the original NFAs were equivalent. A drawback of this approach, is that one first has to determinize the automata before one can check their equivalence. In general, it would be more efficient to check for equivalence during the determinization, 'on-the-fly' that is. As soon as any fact in contradiction with equivalence is detected, the determinization can be aborted. In most cases this results in a fewer number of computation steps. In the worst case, i.e. the two programs are equivalent, one still has to determinize the complete automata.

4.1 Algorithm

This on-the-fly determinization algorithm is based on the notion of bisimulation. A bisimulation is a relation $R \subseteq \mathcal{P}(Q) \times \mathcal{P}(Q)$. Two sets of states X and Y are bisimilar if $o^\sharp(X) = o^\sharp(Y)$ and for all $\alpha \in At$, $p \in \Sigma$, it holds that $(t^\sharp_{\alpha p}(X), t^\sharp_{\alpha p}(Y)) \in R$. $o^\sharp$ maps a set of states to their immediate output. $t^\sharp_{\alpha p}$ maps a

set of states to the set of output states from the transition (α, p).

Definition 4.1. *For each $X \subseteq Q$, the immediate output $o^\sharp$ is defined as*

$$o^\sharp(X) = \begin{cases} 0 & \text{if } X = \emptyset \\ \mathtt{out}(x) & \text{if } X = \{x\} \\ o^\sharp(X_1) + o^\sharp(X_2) & \text{otherwise, } X = X_1 \cup X_2 \end{cases}$$

Definition 4.2. *For each $X \subseteq Q$, $\alpha \in At$, $p \in \Sigma$, the set of output states $t^\sharp_{\alpha p}$ is defined as*

$$t^\sharp_{\alpha p}(X) = \begin{cases} \emptyset & \text{if } X = \emptyset \\ \{d \mid \exists b : (x \xrightarrow{(b,p)} d \in \Delta \wedge \alpha \leq b)\} & \text{if } X = \{x\} \\ t^\sharp_{\alpha p}(X_1) \cup t^\sharp_{\alpha p}(X_2) & \text{otherwise, } X = X_1 \cup X_2 \end{cases}$$

Note that the NFAs constructed in Section 3 have transitions of the form $Exp \times BExp \times \Sigma \times Exp$. In Def. 4.2 transitions of the form $Exp \times At \times \Sigma \times Exp$ are defined instead. This is because the algorithm requires that all transitions are checked. Since $At = 2^T$ and $BExp = 2^{2^T}$ it is more efficient to loop over At than over $BExp$. Either way it is clear that all transitions are checked.

$o^\sharp$ and $t^\sharp_{\alpha p}$ are used to determinize on-the-fly. To check two sets of states for equivalence, their output $o^\sharp$ has to be the same. Then, their respective $t^\sharp_{\alpha p}$'s have to be equivalent for all $\alpha \in At$, $p \in \Sigma$. Subsection 4.2 describes the congruence closure used to avoid redundant computations. It does not take into account only the R established so far, but also the sets of states that are still in the *todo* set.

The full algorithm is outlined below. It was introduced by Pous et al. [3] for KA and we adapted it for KAT. Note that the square bracket notation $Q_{A,B}[0]$ is used on line 4 to indicate that the first argument of the tuple is selected. Also note that for $t^\sharp_{\alpha p}(X')$ and $t^\sharp_{\alpha p}(X')$ it can be decided from X' and Y' whether to use Δ_A or Δ_B.

INPUT: $Q_A, \Delta_A, Q_B, \Delta_B$
OUTPUT: `bool`

```
1  R is empty; todo is empty;
2  insert (Q_A[0], Q_B[0]) in todo;
3  while todo is not empty do
4      extract (X', Y') from todo;
5      if (X', Y') ∉ c(R ∪ todo) then
6          if o♯(X') ≠ o♯(Y') then
7              return false;
8          fi
9          for all α ∈ At do
10             for all p ∈ Σ do
11                 insert (t♯_αp(X'), t♯_αp(Y')) in todo;
12             od
13         od
14         insert (X', Y') in R
15     fi
16 od
17 return true;
```

4.2 Bisimulation up-to congruence

This on-the-fly determinization algorithm for checking equivalence uses a bisimulation up-to congruence method [3] to avoid having to construct the full deterministic automata in case of equivalence. This can be seen on line 5 of the algorithm. It requires a method to check whether two sets of states X and Y

belong to the congruence closure of a relation. The solution of this problem that is introduced by Pous et al. is based on two rewriting rules:

$$X \to X \cup Y \qquad Y \to X \cup Y$$

These rules can be used to compute the normal form of sets of states. Pous et al. use the following definition:

Definition 4.3. *Let $R \subseteq \mathcal{P}(Q) \times \mathcal{P}(Q)$ be a relation on sets of states. We define $\rightsquigarrow_R \subseteq \mathcal{P}(Q) \times \mathcal{P}(Q)$ as the smallest irreflexive relation that satisfies the following rules:*

$$\frac{X \ R \ Y}{X \rightsquigarrow_R X \cup Y} \ (1) \qquad \frac{X \ R \ Y}{Y \rightsquigarrow_R X \cup Y} \ (2) \qquad \frac{Z \rightsquigarrow_R Z'}{U \cup Z \rightsquigarrow_R U \cup Z'} \ (3)$$

Then, for all relations R $\rightsquigarrow_R$ is convergent. We denote by $X \downarrow_R$ the normal form with respect to $\rightsquigarrow_R$. The normal form of a set can be thought of as the largest set of its equivalence class. Pous et al. use the following example.

Let $R = \{(x, u), (y \cup z, u)\}$. The common normal form of $x \cup y$ and u can then be computed.

$$\begin{aligned} x \cup y &\Rightarrow x \cup y \cup u \Rightarrow x \cup y \cup z \cup u \\ u &\Rightarrow x \cup u \qquad \Rightarrow x \cup y \cup z \cup u \end{aligned}$$

The common normal form is then $x \cup y \cup z \cup u$. Now we can define the congruence closure $c(R)$ of R. For all relations R and for all $X, Y \in \mathcal{P}(Q)$, $X \downarrow_R = Y \downarrow_R$ if and only if $(X, Y) \in c(R)$. In the algorithm, we check if $(X, Y) \in c(R \cup todo)$ for some $X, Y \in \mathcal{P}(Q)$. In order to do this, the normal form of X and Y with respect to $\rightsquigarrow_{R \cup todo}$ has to be computed.

5 Example

5.1 Construction

This example is the same as the one given by Broda et al. [9]. Although their construction has the same worst case complexity as ours, this example shows how our construction performs better. It may have less than $||e|| + 1$ states, while Broda et al.'s construction has exactly $||e|| + 1$ states.

Let $T = \{t_1, t_2, t_3\}$, and $\Sigma = \{p, q\}$. Let $e = t_1 p(pq^* t_2 + t_3 q)^*$.

$$\begin{aligned} &\texttt{first}(t_1 p(pq^* t_2 + t_3 q)^*) &&= \{(t_1, p)\} \\ &\texttt{out}(t_1 p(pq^* t_2 + t_3 q)^*) &&= 0 \end{aligned}$$

Now, we execute the algorithm described in Section 4. For each element in the `first` we compute the derivatives. We iterate this process until the derivatives are calculated for the `first` set and no new expressions are found.

$$\begin{aligned} &\delta_{t_1 p}(t_1 p(pq^* t_2 + t_3 q)^*) &&= \{(pq^* t_2 + t_3 q)^*\} \\ &\texttt{first}((pq^* t_2 + t_3 q)^*) &&= \{(1, p), (t_3, q)\} \\ &\texttt{out}((pq^* t_2 + t_3 q)^*) &&= 1 \end{aligned}$$

$$\begin{aligned} &\delta_{1p}((pq^* t_2 + t_3 q)^*) &&= \{q^* t_2 (pq^* t_2 + t_3 q)^*\} \\ &\texttt{first}(q^* t_2 (pq^* t_2 + t_3 q)^*) &&= \{(1, q), (t_2, p), (t_2 t_3, q)\} \\ &\texttt{out}(q^* t_2 (pq^* t_2 + t_3 q)^*) &&= t_2 \end{aligned}$$

$$\delta_{t_3 q}((pq^* t_2 + t_3 q)^*) = \{(pq^* t_2 + t_3 q)^*\}$$

Note that for this expression we have already created a state and calculated its `first`, `out` and derivatives. We do not compute them again.

$$\delta_{1q}(q^*t_2(pq^*t_2 + t_3q)^*) \quad = \{q^*t_2(pq^*t_2 + t_3q)^*\}$$

$$\delta_{t_2p}(q^*t_2(pq^*t_2 + t_3q)^*) \quad = \{q^*t_2(pq^*t_2 + t_3q)^*\}$$

$$\delta_{t_2t_3q}(q^*t_2(pq^*t_2 + t_3q)^*) \quad = \{(pq^*t_2 + t_3q)^*\}$$

Figure 1 shows the NFA that results from the construction method described in Subsection 3.2.

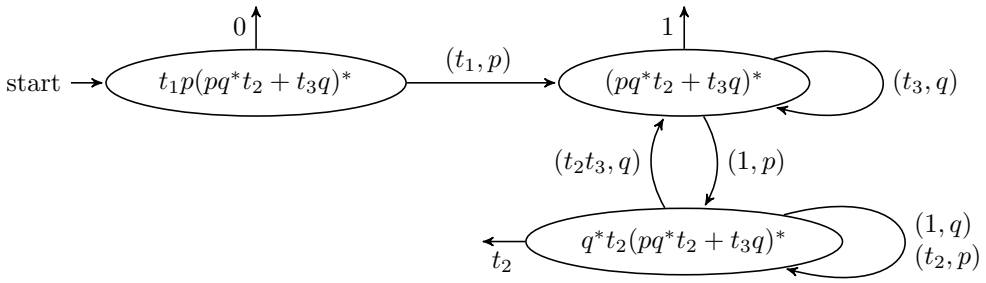

Figure 1: The NFA corresponding to expression $e = t_1p(pq^*t_2 + t_3q)^*$.

5.2 Equivalence

We will use the NFAs from Figure 1 and Figure 2 to compute the equivalence relation. It is clear that these automata, corresponding to $t_1p(pq^*t_2 + t_3q)^*$ and t_1p, respectively, are not equivalent.

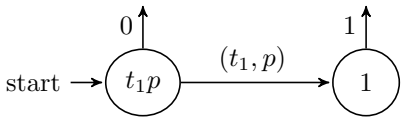

Figure 2: The NFA corresponding to expression $e = t_1p$.

The algorithm is started with $Q_A[0]$ and $Q_B[0]$, which are the sets of states of the respective NFAs. Recall that there are basically two criteria for equivalence of two sets of sets $X, Y \subseteq Q$. Firstly, the immediate output has to be the same. Thus, $o^\sharp(X) = o^\sharp(Y)$. Secondly, the sets of output states for each $At \times \Sigma$ have to be equivalent. Thus, $t^\sharp_{\alpha p}(X)$ and $t^\sharp_{\alpha p}(Y)$ have to be equivalent for all $At \times \Sigma$.

$$\begin{array}{ll} o^\sharp(Q_A[0]) & = 0 + 1 + t_2 \quad = 1 \\ o^\sharp(Q_B[0]) & = 0 + 1 \qquad\quad = 1 \end{array}$$

This satisfies the first criterium. Now we check the equivalence of $t^\sharp_{\alpha p}(X)$ and $t^\sharp_{\alpha p}(Y)$ for each $At \times \Sigma$. Let us say that we start with $\bar{t_1}\bar{t_2}\bar{t_3} \in At$ and $p \in \Sigma$. Then:

$$\begin{array}{ll} t^\sharp_{\bar{t_1}\bar{t_2}\bar{t_3}p}(Q_A[0]) & = \{q^*t_2(pq^*t_2 + t_3q)^*\} \\ t^\sharp_{\bar{t_1}\bar{t_2}\bar{t_3}p}(Q_B[0]) & = \emptyset \end{array}$$

Now we apply the two criteria to $\{q^*t_2(pq^*t_2 + t_3q)^*\}$ and $\emptyset$.

$$\begin{array}{ll} o^\sharp(\{q^*t_2(pq^*t_2 + t_3q)^*\}) & = t_2 \\ o^\sharp(\emptyset) & = 0 \end{array}$$

Since $o^\sharp(\{q^*t_2(pq^*t_2 + t_3q)^*\}) \neq o^\sharp(\emptyset)$, the first criterium is violated. Therefore, we can already conclude that the NFAs from Figure 1 and Figure 2 are not equivalent. Hence, $L(M_1) \neq L(M_2)$ and

$L(t_1p(pq^*t_2 + t_3q)^*) \neq L(t_1p)$. It is clear that this procedure is significantly more efficient than the naive approach that determinizes prior to the equivalence checking.

6 Related work and conclusion

The complexity of earlier finite automaton constructions [6] also depended on the T parts of an expression. Since $At = 2^T$ and $BExp = 2^{2^T}$, this results in very large automata. The complexity of our construction only depends on the Σ parts of the expression. Hence, it has the first finite state automaton to have the same descriptional complexity behavior as its KA counterpart. Other types of automata have been constructed that do have the same complexity behavior, such as the Glushkov and equation automata constructed by Broda et al. [9]. Their Glushkov automaton has exactly $||e|| + 1$ states, while our NFAs have at most $||e|| + 1$ states. The Glushkov automaton has $O(||e||^2)$ transitions in the worst case. For our construction it is exactly $||e||^2$ in the worst case. Similar to our construction, the equation automaton introduced by Broda et al. has at most $||e|| + 1$ states. Alexandra Silva's position automata [12] are generally larger than our automata, since the complexity also depends on the primitive test symbols. However, the construction of the position automata is highly efficiently. It is a trade-off of where one has to pay the price, in the size or the construction of the automata.

Thus, the NFAs resulting from our construction are smaller than any resulting from existing finite state automaton constructions for KAT. They have a size comparable to Glushkov and equation automata that have been constructed for KAT. By working with $BExp \times \Sigma$ transitions instead of $At \times \Sigma$ and by using the `first` set of an expression, many redundant computations are avoided.

There are two ways in which our work can be improved upon. For one, the partial derivatives used in this thesis are not suitable for generic use, since the $BExp$ are not taken into account. This is done as to avoid computational explosion caused by satisfiability. Further research can be done to define partial derivatives of the type δ_{bp} that do address this issue while still avoiding the satisfiability problem. The second improvement might address a stronger bisimilation up-to method to decrease the number of computation steps in the equivalence checking even further, e.g. a bisimulation up-to context.

References

[1] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[2] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.

[3] Damien Pous. Kleene algebra with tests and coq tools for while programs. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, ITP'13, pages 180–196, Berlin, Heidelberg, 2013. Springer-Verlag.

[4] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[5] Dexter Kozen. Automata on guarded strings and applications. *Matématica Contemporânea*, 24:117–139, 2003.

[6] Dexter Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report `http://hdl.handle.net/1813/10173`, Computing and Information Science, Cornell University, March 2008.

[7] Valentin Antimirov. Partial derivatives of regular expressions and finite automata constructions. *Theoretical Computer Science*, 155:291–319, 1995.

[8] Janusz A. Brzozowski. Derivatives of regular expressions. *JOURNAL OF THE ACM*, 11:481–494, 1964.

[9] S. Broda, A. Machiavelo, N. Moreira, and R. Reis. Glushkov and equation automata for KAT expressions. Technical Report DCC-2013-07, Faculdade de Ciências da Universidade do Porto, April 2013.

[10] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959.

[11] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[12] Alexandra Silva. *Kleene Coalgebra*. PhD thesis, Radboud Universiteit Nijmegen, 2010.