Symbolic Regex Matcher

Olli Saarikivi[1], Margus Veanes[1](✉), Tiki Wan[2], and Eric Xu[2]

[1] Microsoft Research, Redmond, USA
margus@microsoft.com
[2] Microsoft Azure, Redmond, USA

Abstract. Symbolic regex matcher is a new open source .NET regular expression matching tool and match generator in the Microsoft Automata framework. It is based on the .NET regex parser in combination with a set based representation of character classes. The main feature of the tool is that the core matching algorithms are based on symbolic derivatives that support extended regular expression operations such as intersection and complement and also support a large set of commonly used features such as bounded loop quantifiers. The particularly useful features of the tool are that it supports full UTF16 encoded strings, the match generation is backtracking free, thread safe, and parallelizes with low overhead in multithreaded applications. We discuss the main design decisions behind the tool, explain the core algorithmic ideas and how the tool works, discuss some practical usage scenarios, and compare it to existing state of the art.

1 Motivation

We present a new tool called *Symbolic Regex Matcher* or *SRM* for fast match generation from extended regular expressions. The development of SRM has been motivated by some concrete industrial use cases and should meet the following *expectations*. Regarding *performance*, the overall algorithm complexity of match generation should be *linear* in the length of the input string. Regarding *expressivity*, it should handle common types of .NET regexes, including support for *bounded quantifiers* and *Unicode categories*; while nonregular features of regexes, such as back-references, are not required. Regarding *semantics*, the tool should be .NET compliant regarding strings and regexes, and the main type of match generation is: *earliest eager nonoverlapping* matches in the input string. Moreover, the tool should be safe to use in distributed and *multi threaded* development environments. Compilation time should be reasonable but it is not a critical factor because the intent is that the regexes are used frequently but updated infrequently. A concrete application of SRM is in an internal tool at Microsoft that scans for credentials and other sensitive content in cloud service software, where the search patterns are stated in form of individual regexes or in certain scenarios as intersections of regexes.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 372–378, 2019.
https://doi.org/10.1007/978-3-030-17462-0_24

The built-in .NET regex engine uses a backtracking based match search algorithm and does not meet the above expectations; in particular, some patterns may cause *exponential* search time. While SRM uses the *same parser* as the .NET regex engine, its back-end is a new engine that is built on the notion of *derivatives* [1], is developed as a tool in the open source Microsoft Automata framework [5], the framework was originally introduced in [8]. SRM meets *all* of the above expectations. Derivatives of regular expressions have been studied before in the context of matching of regular expressions, but only in the functional programming world [2,6] and in related domains [7]. Compared to earlier derivative based matching engines, the new contribution of SRM is that it supports *match generation* not only match detection, it supports extended features, such as *bounded quantifiers*, *Unicode categories*, and *case insensitivity*, it is .NET compliant, and is implemented in an imperative language. As far as we are aware of, SRM is the first tool that supports derivative based match generation for extended regular expressions. In our evaluation SRM shows significant performance improvements over .NET, with more predictable performance than RE2 [3], a state of the art automata based regex matcher.

In order to use SRM in a .NET application instead of the built-in match generator, `Microsoft.Automata.dll` can be built from [5] on a .NET platform version 4.0 or higher. The library extends the built-in `Regex` class with methods that expose SRM, in particular through the `Compile` method.

2 Matching with Derivatives

Here we work with derivatives of *symbolic extended regular expressions* or *regexes* for short. *Symbolic* means that the basic building blocks of single character regexes are *predicates* as opposed to singleton characters. In the case of standard .NET regexes, these are called *character classes*, such as the class of digits or `\d`. In general, such predicates are drawn from a given effective Boolean algebra and are here denoted generally by α and β; $\bot$ denotes the *false* predicate and `.` the *true* predicate. For example, in .NET $\bot$ can be represented by the empty character class `[0-[0]]`.[1] *Extended* here means that we allow *intersection*, *complement*, and *bounded quantifiers*.

The abstract syntax of regexes assumed here is the following, assuming the usual semantics where $()$ denotes the empty sequence ϵ and $\langle\alpha\rangle$ denotes any singleton sequence of character that belongs to the set $[\![\alpha]\!] \subseteq \Sigma$, where Σ is the alphabet, and n and m are nonnegative integers such that $n \leq m$:

$$() \quad \langle\alpha\rangle \quad R_1R_2 \quad R\{n,m\} \quad R\{n,*\} \quad R_1|R_2 \quad R_1\&R_2 \quad \neg R$$

where $[\![R\{n,m\}]\!] \stackrel{\text{def}}{=} \{v \in [\![R]\!]^i \mid n{\leq}i{\leq}m\}$, $[\![R\{n,*\}]\!] \stackrel{\text{def}}{=} \{v \in [\![R]\!]^i \mid n{\leq}i\}$. The expression $R*$ is a shorthand for $R\{0,*\}$. We write $\bot$ also for $\langle\bot\rangle$. We assume that $[\![R_1|R_2]\!] = [\![R_1]\!] \cup [\![R_2]\!]$, $[\![R_1\&R_2]\!] = [\![R_1]\!] \cap [\![R_2]\!]$, and $[\![\neg R]\!] = \Sigma^* \setminus [\![R]\!]$.

[1] The more intuitive syntax `[]` is unfortunately not allowed.

A less known feature of the .NET regex grammar is that it also supports *if-then-else* expressions over regexes, so, when combined appropriately with $\bot$ and $\textbf{.}$, it also supports intersection and complement. R is *nullable* if $\epsilon \in [\![R]\!]$. Nullability is defined recursively, e.g., $R\{n,m\}$ is nullable iff R is nullable or $n = 0$.

Given a concrete character x in the underlying alphabet Σ, and a regex R, the *x-derivative of* R, denoted by $\partial_x R$, is defined on the right. Given a language $L \subseteq \Sigma^*$, the x-derivative of L, $\boldsymbol{\partial}_x L \stackrel{\text{def}}{=} \{v \mid xv \in L\}$. It is well-known that $[\![\partial_x R]\!] = \boldsymbol{\partial}_x [\![R]\!]$. The abstract derivation rules provide a way to decide if an input u matches a regex R as follows. If $u = \epsilon$ then u matches R iff R is nullable; else, if $u = xv$ for some $x \in \Sigma, v \in \Sigma^*$ then u matches R iff v matches $\partial_x R$. In other words, the derivation rules can be unfolded lazily to create the transitions of the underlying DFA. In this setting we are considering Brzozowski derivatives [1].

$$\begin{aligned}
\partial_x() &\stackrel{\text{def}}{=} \bot \\
\partial_x\langle\alpha\rangle &\stackrel{\text{def}}{=} \begin{cases} (), & \text{if } x \in [\![\alpha]\!]; \\ \bot, & \text{otherwise.} \end{cases} \\
\partial_x(R_1R_2) &\stackrel{\text{def}}{=} \begin{cases} ((\partial_x R_1)R_2)|\partial_x R_2, & \text{if } R_1 \text{ is nullable;} \\ (\partial_x R_1)R_2, & \text{otherwise.} \end{cases} \\
\partial_x R\{n,m\} &\stackrel{\text{def}}{=} \begin{cases} (\partial_x R)R\{n{-}1,m{-}1\}, & \text{if } n{>}0; \\ (\partial_x R)R\{0,m{-}1\}, & \text{if } n{=}0 \text{ and } m{>}0; \\ \bot, & \text{otherwise (since } R\{0,0\} \stackrel{\text{def}}{=} ()). \end{cases} \\
\partial_x R\{n,*\} &\stackrel{\text{def}}{=} \begin{cases} (\partial_x R)R\{n-1,*\}, & \text{if } n > 0; \\ (\partial_x R)R\{0,*\}, & \text{otherwise.} \end{cases} \\
\partial_x(R_1|R_2) &\stackrel{\text{def}}{=} (\partial_x R_1)|(\partial_x R_2) \\
\partial_x(R_1\&R_2) &\stackrel{\text{def}}{=} (\partial_x R_1)\&(\partial_x R_2) \\
\partial_x\neg R &\stackrel{\text{def}}{=} \neg\partial_x R
\end{aligned}$$

Match Generation. The main purpose of the tool is to *generate* matches. While match generation is a topic that has been studied extensively for classical regular expressions, we are not aware of efforts that have considered the use of derivatives and extended regular expressions in this context, while staying *backtracking free* in order to guarantee *linear complexity* in terms of the length of the input. Our matcher implements by default *nonoverlapping earliest eager* match semantics. An important property in the matcher is that the above set of regular expressions is closed under *reversal*. The reversal of regex R is denoted $R^{\mathbf{r}}$. Observe that:

$$(R_1R_2)^{\mathbf{r}} \stackrel{\text{def}}{=} (R_2^{\mathbf{r}}R_1^{\mathbf{r}}) \quad R\{n,m\}^{\mathbf{r}} \stackrel{\text{def}}{=} R^{\mathbf{r}}\{n,m\} \quad R\{n,*\}^{\mathbf{r}} \stackrel{\text{def}}{=} R^{\mathbf{r}}\{n,*\}$$

It follows that $[\![R^{\mathbf{r}}]\!] = [\![R]\!]^{\mathbf{r}}$ where $L^{\mathbf{r}}$ denotes the reversal of $L \subseteq \Sigma^*$. The match generation algorithm can now be described at a high level as follows. Given a regex R, find all the (nonoverlapping earliest eager) matches in a given input string u. This procedure uses the three regexes: R, $R^{\mathbf{r}}$ and $\textbf{.}*R$:

1. Initially $i = 0$ is the start position of the first symbol u_0 of u.
2. Let $i_{\text{orig}} = i$. Find the *earliest* match starting from i and $q = \textbf{.}*R$: Compute $q := \partial_{u_i} q$ and $i := i + 1$ until q is nullable. **Terminate** if no such q exists.
3. Find the *start position* for the above match closest to i_{orig}: Let $p = R^{\mathbf{r}}$. While $i > i_{\text{orig}}$ let $p := \partial_{u_i} p$ and $i := i - 1$, if p is nullable let $i_{\text{start}} := i$.

4. Find the *end position* for the match: Let $q = R$ and $i = i_{\text{start}}$. Compute $q := \partial_{u_i} q$ and $i := i + 1$ and let $i_{\text{end}} := i$ if q is nullable; repeat until $q = \bot$.
5. **Return** the match from i_{start} to i_{end}.
6. Repeat step 2 from $i := i_{\text{end}} + 1$ for the next *nonoverlapping* start position.

Observe that step 4 guarantees *longest* match in R from the position i_{start} found in step 3 for the earliest match found in step 2. In order for the above procedure to be practical there are several optimizations that are required. We discuss some of the implementation aspects next.

3 Implementation

SRM is implemented in `C#`. The input to the tool is a .NET regex (or an array of regexes) that is compiled into a serializable object R that implements the main matching interface `IMatcher`. Initially, this process uses a Binary Decision Diagram (*BDD*) based representation of predicates in order to efficiently canonicalize various conditions such as *case insensitivity* and *Unicode categories*. The use of BDDs as character predicates is explained in [4]. Then all the BDDs that occur in R are collected and their *minterms* (satisfiable Boolean combinations) are calculated, called the *atoms* $(\alpha_1, \ldots, \alpha_k)$ *of* R, where $\{[\![\alpha_i]\!]_{\texttt{BDD}}\}_{i=1}^{k}$ forms a partition of Σ. Each `BDD`-predicate α in R is now translated into a k-bit bit-vector (or `BV`) value β whose i'th bit is 1 iff $\alpha \wedge_{\texttt{BDD}} \alpha_i$ is nonempty. Typically k is small (often $k \leq 64$) and allows `BV` to be implemented very efficiently (often by `ulong`), where $\wedge_{\texttt{BV}}$ is bit-wise-and. All subsequent Boolean operations are performed on this more efficient and *thread safe* data type. The additional step required during input processing is that each concrete input character c (`char` value) is now first mapped into an atom id i that determines the bit position in the `BV` predicate. In other words, $c \in [\![\beta]\!]_{\texttt{BV}}$ is implemented by finding the index i such that $c \in [\![\alpha_i]\!]_{\texttt{BDD}}$ and testing if the i'th bit of β is 1, where the former search is hardcoded into a precomputed lookup table or decision tree.

For example let R be constructed for the regex `\w\d*`. Then R has three atoms: $[\![\alpha_1]\!] = \Sigma \setminus [\![\texttt{\\w}]\!]$, $[\![\alpha_2]\!] = [\![\texttt{\\d}]\!]$, and $[\![\alpha_3]\!] = [\![\texttt{\\w}]\!] \setminus [\![\texttt{\\d}]\!]$, since $[\![\texttt{\\d}]\!] \subset [\![\texttt{\\w}]\!]$. For example `BV` 110_2 represents `\w` and 010_2 represents `\d`.

The symbolic regex AST type is treated as a value type and is handled similarly to the case of derivative based matching in the context of functional languages [2,6]. A key difference though, is that *weak equivalence* [6] checking is not enough to avoid state-space explosion when *bounded quantifiers* are allowed. A common situation during derivation is appearance of subexpressions of the form $(A\{0,k\}B)|(A\{0,k-1\}B)$ that, when kept unchecked, keep reintroducing disjuncts of the same subexpression but with smaller value of the upper bound, potentially causing a substantial blowup. However, we know that $A\{0,n\}B$ is *subsumed* by $A\{0,m\}B$ when $n \leq m$, thus $(A\{0,m\}B)|(A\{0,n\}B)$ can be simplified to $A\{0,m\}B$. To this end, a disjunct $A\{0,k\}B$, where $k > 0$, is represented internally as a *multiset element* $\langle A, B\rangle \mapsto k$ and the expression $(\langle A, B\rangle \mapsto m)|(\langle A, B\rangle \mapsto n)$ reduces to $(\langle A, B\rangle \mapsto \max(m, n))$. This is a form of *weak subsumption checking* that provides a crucial optimization step during derivation. Similarly, when A and B are both singletons, say $\langle\alpha\rangle$ and $\langle\beta\rangle$, then $\langle\alpha\rangle|\langle\beta\rangle$ reduces to $\langle\alpha \vee_{\mathtt{BV}} \beta\rangle$ and $\langle\alpha\rangle\&\langle\beta\rangle$ reduces to $\langle\alpha \wedge_{\mathtt{BV}} \beta\rangle$. Here thread safety of the Boolean operations is important in a multi threaded application.

Finally, two more key optimizations are worth mentioning. First, during the main match generation loop, symbolic regex nodes are internalized into integer state ids and a DFA is maintained in form of an integer array δ indexed by $[i, q]$ where $1 \leq i \leq k$ is an atom index, and q is a state integer id, such that old state ids are immediately looked up as $\delta[i, q]$ and not rederived. Second, during step 2, initial search for the *relevant initial prefix*, when applicable, is performed using `string.IndexOf` to completely avoid the trivial initial state transition corresponding to the loop $\partial_c\mathtt{.}{*}R = \mathtt{.}{*}R$ in the case when $\partial_c R = \bot$.

4 Evaluation

We have evaluated the performance of SRM on two benchmarks:

Twain: 15 regexes matched against a 16 MB file containing the collected works of Mark Twain.

Assorted: 735 regexes matched against a synthetic input that includes some matches for each regex concatenated with random strings to produce an input file of 32 MB. The regexes are from the Automata library's samples and were originally collected from an online database of regular expressions.

We compare the performance of our matcher against the built-in .NET regex engine and Google's RE2 [3], a state of the art backtracking free regex match generation engine. RE2 is written in `C++` and internally based on automata. It eliminates bounded quantifiers in a preprocessing step by unwinding them, which may cause the regex to be rejected if the unwinding exceeds a certain limit. RE2 does not support extended operations over regexes such as intersection or complement. We use RE2 through a `C#` wrapper library.

The input to the built-in .NET regex engine and SRM is in UTF16, which is the encoding for NET's built-in strings, while RE2 is called with UTF8 encoded input. This implies for example that a regex such as `[\uD800-\uDFFF]` that tries to locate a single UTF16 surrogate is not meaningful in the context UTF8. All experiments were run on a machine with dual Intel Xeon E5-2620v3 CPUs running Windows 10 with .NET Framework 4.7.1. The reported running times for **Twain** are averages of 10 samples, while the statistics for **Assorted** are based on a single sample for each regex.

Figure 1 presents running times for each regex in **Twain**, while Fig. 2 presents a selection of metrics for the **Assorted** benchmark.

Both SRM and RE2 are faster than .NET on most regexes. This highlights the advantages of automata based regular expression matching when the richer features of a backtracking matcher are not required.

Compilation of regular expressions into matcher objects takes more time in SRM than RE2 or .NET. The largest contributor to this is finding the minterms of all predicates in the regex. For use cases where initialization time is critical

Regex	.NET	RE2	SRM
`Twain`	0.20s	**0.02s**	0.05s
`(?i)Twain`	0.66s	**0.26s**	0.39s
`[a-z]shing`	4.11s	**0.21s**	0.78s
`Huck[a-zA-Z]+|Saw[a-zA-Z]+`	1.47s	0.21s	**0.11s**
`[a-q][^u-z]{13}x`	10.20s	16.64s	**3.07s**
`Tom|Sawyer|Huckleberry|Finn`	1.53s	0.24s	**0.12s**
`(?i)Tom|Sawyer|Huckleberry|Finn`	6.73s	**0.22s**	0.51s
`.{0,2}(Tom|Sawyer|Huckleberry|Finn)`	15.84s	**0.22s**	0.64s
`.{2,4}(Tom|Sawyer|Huckleberry|Finn)`	16.15s	**0.22s**	0.62s
`Tom.{10,25}river|river.{10,25}Tom`	1.83s	**0.21s**	**0.21s**
`[a-zA-Z]+ing`	9.62s	**0.73s**	0.92s
`\s[a-zA-Z]{0,12}ing\s`	5.56s	**0.30s**	0.82s
`([A-Za-z]awyer|[A-Za-z]inn)\s`	6.26s	**0.21s**	0.87s
`["’][^"’]{0,30}[?!\.][\"’]`	2.00s	0.25s	**0.19s**
`\p{Sm}`	1.71s	0.21s	**0.10s**

Fig. 1. Time to generate all matches for each regex in **Twain**.

Metric	.NET	RE2	SRM
# of regexes with best time	0	305	**430**
Total compilation time for all regexes	0.8s	**0.1s**	13.9s
Average matching time	7.93s	0.36s	**0.25s**
80th percentile matching time	2.24s	**0.08s**	0.13s

Fig. 2. Metrics for the **Assorted** benchmark.

and inputs are known in advance, SRM provides support for pre-compilation and fast deserialization of matchers.

Comparing SRM to RE2 we can see that both matchers have regexes they do better on. While SRM achieves a lower average matching time on **Assorted**, this is due to the more severe outliers in RE2's performance profile, as shown by the lower 80th percentile matching time. Overall SRM offers performance that is comparable to RE2 while being implemented in `C#` without any unsafe code.

Application to Security Leak Scanning. SRM has been adopted in an internal tool at Microsoft that scans for credentials and other sensitive content in cloud service software. With the built-in .NET regex engine the tool was susceptible to catastrophic backtracking on files with long lines, such as minified JavaScript and SQL server seeding files. SRM's linear matching complexity has helped address these issues, while maintaining compatibility for the large set of .NET regexes used in the application.

References

1. Brzozowski, J.A.: Derivatives of regular expressions. JACM **11**, 481–494 (1964)
2. Fischer, S., Huch, F., Wilke, T.: A play on regular expressions: functional pearl. SIGPLAN Not. **45**(9), 357–368 (2010)
3. Google: RE2. https://github.com/google/re2

4. Hooimeijer, P., Veanes, M.: An evaluation of automata algorithms for string analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_18
5. Microsoft: Automata. https://github.com/AutomataDotNet/
6. Owens, S., Reppy, J., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. **19**(2), 173–190 (2009)
7. Traytel, D., Nipkow, T.: Verified decision procedures for MSO on words based on derivatives of regular expressions. SIGPLAN Not. **48**(9), 3–12 (2013)
8. Veanes, M., Bjørner, N.: Symbolic automata: the toolkit. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 472–477. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_33