Adding regular expression operators to OCL

K. Lano[1]

[1]Dept. of Informatics, King's College London, UK

Abstract

In this short paper we provide a systematic proposal to add regular expression operators to the OCL standard library for the String type, and we discuss other String operators which could be included in the library.

Keywords

Object Constraint Language, OCL, Regular expressions

1. Introduction

The String type is a central data type in most forms of software, however the String type facilities in the OCL 2.4 standard are quite restricted (only 17 specific operators are defined) [9]. One category of omitted operations are those concerning substitution within strings and string matching against regular expressions. Furthermore, since strings can be regarded as sequences of characters (strings of size 1), it would seem natural to adopt sequence operators such as *first*, *last*, *front*, *tail* and *insertAt* for strings. We consider the addition of regular expression operators in Section 2, and the addition of other operators in Section 3. Implementation of the operators is discussed in Section 4, and performance evaluation in Section 6.

2. Regular expressions

Regular expressions are an important facility in many software systems, especially for validation of textual data, data filtering and transformation. We found that these facilities were of key importance in the development of software engineering tools for MDE, in particular our code-generator DSL, $\mathcal{CSTL}$ [7] depends heavily on string pattern matching and replacement. It would be desirable to be able to specify such tools using OCL.

Most mainstream programming languages provide regular expression facilities, but there are considerable differences in the level of support and operations provided (eg., Java $str.matches(patt)$ requires that the entire *str* is a single match for *patt*, whilst Python $patt.match(str)$ only requires matching of *patt* to a *substring* of *str*, and C# $patt.Matches(str)$ returns all matches of *patt* within *str*). Specification languages such as EOL and ATL's OCL version also provide some regular expression facilities [5, 2], but again these languages do not use a consistent terminology.

OCL 2021: 20th International Workshop on OCL and Textual Modeling, June 25 2021, Bergen, Norway
kevin.lano@kcl.ac.uk (K. Lano)

We propose a cohesive set of String operators to enable the use of regular expressions in OCL-based specifications. These operators clearly distinguish between different forms of matching and replacement:

- $isMatch(patt : String) : Boolean$ – returns *true* only if the entire text of *self* is a single match for the regular expression *patt*.
- $hasMatch(patt : String) : Boolean$ – returns *true* if some non-empty substring of *self* satisfies the regular expression *patt*.
- $allMatches(patt : String) : Sequence(String)$ – returns the sequence of non-overlapping and non-empty substrings of *self* which match *patt*, in the order of their occurrence in *self*.
- $replaceAllMatches(patt : String, rep : String) : String$ – A copy of *self* with each substring in $allMatches(patt)$ replaced by *rep*.
- $split(patt : String) : Sequence(String)$ – the sequence of substrings of *self* formed by removing the substrings in $allMatches(patt)$.

For additional clarity, *split* could alternatively be named *splitOnMatches*. Useful auxiliary operations are

- $firstMatch(patt : String) : String$
- $replaceFirstMatch(patt : String, rep : String) : String$

It is also useful to support replacement *substituteFirst* and *substituteAll* of substrings based on literal string equality, as in [4].

Because of the general utility of such operations, we consider that they should be included into the OCL standard library, rather than in a specialised library.

2.1. Regular expression semantics

We assume a core regular expression language, such as POSIX ERE, consisting of the standard metacharacters ., $*$, $+$, ?, [,], |, literal characters, character ranges $c1-c2$, and escaped special characters $\backslash\backslash s$. This language is supported by all mainstream programming languages. It has a semantics given in terms of sets of sequences of characters [6]: $[\![r]\!]$ is the set of sequences of characters which conform to regular expression r. This semantics can be defined by recursion on the structure of r. Eg., $[\![r_1 \mid r_2]\!] = [\![r_1]\!] \cup [\![r_2]\!]$ [6].

We assume that some matching relation $str \models r$ exists between literal strings and regular expressions, this means that $str.characters() \in [\![r]\!]$. For example, $\text{“}Augustus\text{”} \models \text{“}[A-Z][a-z]+\text{”}$ and $not(\text{“}++**\text{”} \models \text{“}[A-Z][a-z]+\text{”})$. $str \models patt$ means that the entire string *str* satisfies *patt*. Thus

$$\text{“}adam@eden.com\text{”} \models \text{“}[A-Za-z]+@[A-Za-z]+\backslash\backslash.[A-Za-z]+\text{”}$$

but $not(\text{“}C++\text{”} \models \text{“}[A-Z]+\text{”})$.

isMatch can then be defined directly in terms of $\models$. $str.isMatch(patt)$ is *invalid* if *str* or *patt* is *invalid*, otherwise it is *null* if either *str* or *patt* is *null*. For valid arguments we define:

```
isMatch(patt : String) : Boolean
pre: true
post:
  if size() = 0 then result = false
  else if patt.size() = 0 then result = false
  else if str |= patt then result = true
  else result = false
  endif endif endif
```

This definition provides a formal definition of *isMatch*, suitable for inclusion in Section A.2.1.3 of [9].

The other regular expression operators can then be defined in terms of *isMatch*. *hasMatch* is:

```
hasMatch(patt : String) : Boolean
pre: true
post:
  result =
    Sequence{1..size()}->exists( x, y |
         x <= y  and  str.substring(x,y).isMatch(patt) )
```

We assume that $str.substring(i, j)$ is “” if $j < i$[1].

$firstMatch(patt : String)$ is defined by:

```
firstMatch(patt : String) : String
pre: true
post:
  if not(hasMatch(patt)) then result = null
  else if
    Sequence{1..size()}->exists( y | substring(1,y).isMatch(patt) )
  then
    result = Sequence{1..size()}->collect( y | substring(1,y) )
               ->select( str | str.isMatch(patt) )->max()
  else
    result = substring(2,size()).firstMatch(patt)
  endif endif
```

The use of *max* here means that we take the largest match of *patt* (the longest matching substring of *self*) starting from a given index. This is the most common matching strategy for regular expressions (‘greedy’ matching), but alternative strategies can be specified analogously.

$str.allMatches(patt)$ is also defined recursively:

```
allMatches(patt : String) : Sequence(String)
pre: true
post:
  if size() = 0 then result = Sequence{}
```

[1]This case is not defined in [9].

```
  else if not(self.hasMatch(patt)) then result = Sequence{}
  else
    let m : String = self.firstMatch(patt) in
      result = Sequence{ m }->union(
        self.substring(
            self.indexOf(m)+m.size(),self.size()).allMatches(patt))
  endif endif
```

Matches cannot be the empty string, by definition of *isMatch*, therefore this recursion is well-defined. An example is $\text{“}abracadabra\text{”}.allMatches(\text{“}ab\text{”}) = Sequence\{\text{“}ab\text{”}, \text{“}ab\text{”}\}$.

str.*replaceAllMatches*(*patt*, *rep*) is defined by a similar recursion. It is *invalid* if any of *str*, *patt* or *rep* are *invalid*. If *patt* or *rep* or *str* are null, the result is *str*.

```
replaceAllMatches(patt : String, rep : String) : String
pre: true
post:
  if patt.size() = 0 or self.size() = 0
  then result = self
  else if not(str.hasMatch(patt)) then result = self
  else
    let m : String = self.firstMatch(patt)
    in
      result =
        substring(1,indexOf(m)-1) + rep +
            substring(indexOf(m)+m.size(),size()).
                    replaceAllMatches(patt,rep)
  endif endif
```

An example is $\text{“}abracadabra\text{”}.replaceAllMatches(\text{“}ab\text{”}, \text{“”}) = \text{“}racadra\text{”}$.

replaceFirstMatch is:

```
replaceFirstMatch(patt : String, rep : String) : String
pre: true
post:
  if patt.size() = 0 or self.size() = 0
  then result = self
  else if not(str.hasMatch(patt)) then result = self
  else
    let m : String = self.firstMatch(patt)
    in
      result =
        substring(1,indexOf(m)-1) + rep +
            substring(indexOf(m)+m.size(),size())
  endif endif
```

str.*split*(*patt*) is defined similarly. It is *invalid* if either *str* or *patt* are *invalid*, and *Sequence*{*str*} if either are null.

```
split(patt : String) : Sequence(String)
pre: true
post:
  if self.size() = 0 or patt.size() = 0
  then result = Sequence{self}
  else if not(hasMatch(patt)) then result = Sequence{self}
  else
    let m : String = self.firstMatch(patt)
    in let ind : Integer = indexOf(m) in
      if ind > 1 then
        result = Sequence{substring(1,ind-1)}->union(
            substring(ind+m.size(),size()).split(patt))
      else
        result = substring(ind+m.size(),size()).split(patt)
  endif endif endif
```

An example is $\text{“}abracadabra\text{”}.split(\text{“}ab\text{”}) = Sequence\{\text{“}racad\text{”}, \text{“}ra\text{”}\}$.

An alternative to using *isMatch* as the core regex operator would be to use *firstMatch*: both *isMatch* and *hasMatch* can be derived from *firstMatch*.

3. Other String operators

Since strings can also be regarded as sequences (of characters), it is often convenient to apply some sequence operators to strings. The $size()$ and $at(i)$ operators are already common to both types in [9], as is $indexOf(str)$, but with a more general meaning for String than for Sequence (the corresponding Sequence operator would be $indexOfSubSequence(sq)$). In addition, the subrange operators $substring(i, j)$ and $subSequence(i, j)$ correspond. In several programming languages strings are treated as pseudo-collections, eg., iterators can be defined over strings in C++ and in Swift, and Python uses the same range selection operators for strings and sequences.

We suggest that $first()$, $last()$, $front()$ and $tail()$ are natural convenience operators to adopt for String. These can be directly defined in terms of *substring*, so do not require extension of Annex A of [9]. $reverse()$ and $insertAt(i, str)$ would also be relevant operators to adopt, where *insertAt* allows a general string to be inserted into another:

$$s.insertAt(i, str) \mathrel{\widehat{=}} s.substring(1, i-1) + str + s.substring(i, s.size())$$

Other widely-used string operators are $trim()$ to remove leading and trailing whitespace, $lastIndexOf(str)$, $hasPrefix(str)/startsWith(str)$, $hasSuffix(str)/endsWith(str)$ and $after(str)$ and $before(str)$. Thus we also recommend these for inclusion in the OCL String library.

However there are many specialised string operators which are more appropriately defined in external extension libraries. For example, computation of the edit distance of two strings [8] or the generation of random strings.

4. Operator implementations

The operators *isMatch*, *hasMatch*, *firstMatch*, *allMatches*, *replaceFirstMatch*, *replaceAllMatches* and *split* have been incorporated into the AgileUML toolset [3] OCL-based specification language and included in the specification analysis facilities of the toolset. Thus they can be used in any specification of business, mobile or web-service systems defined using the tools.

Code-generation for specifications using the regular expression operators is supported by extended OCL libraries for Java 4, 6, 7 and 8, C#, C++, C, Python and Swift, available from [1]. Eg., *Ocl.swift* supports Swift versions 4 and 5. In each library we have defined specific implementations of the operators. For Java we use the java.util.regex library facilities. For C# we use the System.Text.RegularExpressions library. For C++ we use the C++ 2011 <regex.h> library. For ANSI C we use the lcc regex.h library. For Swift we use the NSRegularExpression facilities from Objective-C. For Python we use the *re* library facilities.

In some cases, such as Java, the programming languages directly provide the required operators. In other cases (eg., C, Swift), the operators need to be implemented in terms of more basic matching facilities. We have primarily used iterative algorithms for efficiency.

We provide implementations of $\mathit{first}()$, $\mathit{last}()$, $\mathit{front}()$, $\mathit{tail}()$, $\mathit{reverse}()$, $\mathit{insertAt}(i, \mathit{str})$, $\mathit{trim}()$, $\mathit{lastIndexOf}(\mathit{str})$, $\mathit{after}(\mathit{str})$ and $\mathit{before}(\mathit{str})$ String operators in the AgileUML OCL implementations for Java, C#, C++, C, Python and Swift.

Extension operators such as edit distance are instead defined as static operations of a *StringLib* component. In specifications these are invoked as $\mathit{StringLib.editDist}(s1, s2)$. Implementations of *StringLib* need to be provided in each target programming language. More advanced regular expression facilities, such as first-class patterns and match groups, could also be provided in an extension component.

5. Related work

The OCL 2.4 library [9] provides 17 operations on String, but omits regular expressions and convenience operators such as *hasPrefix*, *hasSuffix*, *lastIndexOf*, *reverse*, *trim*.

The Eclipse OCL [4] library for String additionally provides a *matches* operation (for *isMatch*), *replaceAll*, *replaceFirst*, but not *firstMatch*, *allMatches*, *hasMatch* or *split*.

EOL [5] provides 20 String operators based on Java, including the regular expression operators *matches* (representing *hasMatch*) and *split*, but omits *allMatches*, *replaceAllMatches* and *isMatch*.

ATL OCL [2] provides the OCL 2.4 String operators, additional String operators *trim*, *lastIndexOf*, *startsWith*, *endsWith*, a limited form *replaceAll* of literal string replacement, and regular expression operators *split*, *regexReplaceAll*, but not *isMatch*, *hasMatch* or *allMatches*.

6. Evaluation

In order to evaluate the performance of the regular expression implementations in different languages we defined examples for (i) *allMatches* and (ii) *replaceAllMatches* with the regular expression $[a{-}zA{-}Z]+$ applied to a string of length 1000 containing 100 words in case (i) and expression $\backslash\backslash([\frown\backslash\backslash(\backslash\backslash)] * \backslash\backslash)$ applied to a string of length 500 in (ii). The latter expression matches against ()-bracketed texts which contain no (or) brackets within them. The examples are in oclregexExamples.zip at [1]. Table 1 shows the performance for Java, Python and C# implementations of these examples, iterated for N=100, 1000, 10000 and 100000 times. The results are the averages of three independent executions, with all times in ms. All tests were carried out on a Windows 10 i5-6300 dual core laptop with 2.4GHz clock frequency, 8GB RAM and 3MB cache.

N	*allMatches*			*replaceAllMatches*		
	Java	C#	Python	Java	C#	Python
100	20	16.3	0	0	14	0
1000	63	87.7	21.2	15	23	0
10000	142	354.7	173.6	110	98.7	19.3
100000	890	2,995	1,680.1	219	709	193.2

Table 1
Execution times of *allMatches* and *replaceAllMatches*

This means that for Java each match takes about $10^{-4}ms$ and each replacement about $4 * 10^{-4}ms$ in the largest case. Performance figures are similar for other languages.

7. Summary

In this paper we have provided a rationale for adding a core set of regular expression facilities to OCL, we also described their semantics and how the operators can be translated into different programming language implementations. We also discussed the possible inclusion of other widely-used string operators.

References

[1] AgileUML repository, https://github.com/eclipse/agileuml/, 2021.
[2] Eclipse, *ATL user guide*, eclipse.org, 2019.
[3] Eclipse AgileUML project, https://projects.eclipse.org/projects/modeling.agileuml, 2021.
[4] Eclipse OCL Version 6.4.0, https://projects.eclipse.org/projects/modeling.mdt.ocl, 2021.
[5] The Epsilon Object Language, https://www.eclipse.org/epsilon/doc/eol, 2020.
[6] D. Gries, *Compiler construction for digital computers*, Wiley, 1971.
[7] K. Lano, Q. Xue, S. Kolahdouz-Rahimi, *Agile specification of code generators for model-driven engineering*, ICSEA 2020.
[8] I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Cybernetics and control theory, vol. 10 (8), 1966, pp. 707–710.
[9] OMG, *Object Constraint Language 2.4 Specification*, OMG document formal/2014-02-03, 2014.