# Model and implementation of unfolding iCalendar documents

Roger L. Costello  
July 17, 2017

**iCalendar** is a computer file format which allows Internet users to send meeting requests and tasks to other Internet users by sharing or sending files in this format. [[Wikipedia](https://en.wikipedia.org/wiki/ICalendar)]

An iCalendar file consists of a series of lines. Each line cannot be longer than 75 characters (actually 75 octets, but that’s irrelevant for this discussion). If a line exceeds that length, the line must be “folded” onto the next line. A space character at the start of a line indicates that it is a continuation of the previous line.

The process of moving from the folded representation to its single-line representation is called "unfolding". Unfolding is accomplished by removing the CRLF and the space character that immediately follows … *When parsing a content line, folded lines must first be unfolded*. [[RFC5545](https://tools.ietf.org/html/rfc5545#section-3.1)]

Here is a sample iCalendar file:

BEGIN:VCALENDAR  
VERSION:2.0  
CALSCALE:GREGORIAN  
BEGIN:VEVENT  
DESCRIPTION;ALTREP="cid:part1.0001@example.org":The Fall'98 Wild  
 Wizards Conference - - Las Vegas\, NV\, USA  
END:VEVENT  
END:VCALENDAR

The value of the DESCRIPTION property is folded onto the next line, as indicated by the space at the start of Wizards Conference ...

This paper first shows an Alloy model of unfolding lines in iCalendar documents. Following the Alloy model is an implementation of unfolding in a programming language. Lastly, I provide some lessons learned.

## Alloy model

These are the key abstractions I identified in the unfolding problem: documents, lines, space, data, and unfolding. In particular, there is a document containing lines, each line has data, if a line starts with a space then it is a continuation line and must be merged with the previous line, and an unfolding operation must be repetitively applied to the document. The resulting document must have no folded lines.

An ordered set of documents represents the states of the document during the unfolding process:

**open** util/ordering[Document]

The document contains a set of lines. Each line has a next line (except the last line, of course). Each line has data. After unfolding, a line will have its original data plus the next line’s data. Thus, a line has a “set” of data. One of the lines, obviously, is the first.

**sig** Document {  
 lines: Line -> **lone** Line,  
 content: Line -> **set** Data,  
 firstLine: Line  
} {  
 firstLine **in** lines.Line  
 // No Line maps to firstLine, i.e., firstLine is the first line  
 **no** line: Line | lines[line] = firstLine  
 // All lines are reachable from firstLine  
 (lines.Line + lines[Line] - firstLine) **in** firstLine.^lines  
 // The lines are sequential, i.e., lines are acyclic  
 **no** ^lines & iden  
 // No space at start of first line  
 **no** firstLine.spaceAtStart  
}

A Line may have a space at its start.

**sig** Line {  
 spaceAtStart: **lone** Space  
}

Space is represented by a singleton set. Data is a set of values.

**one** **sig** Space {}  
**sig** Data {}

Each time the unfold operation is called, it unfolds one line in the document.

Example of an Unfold Operation: Suppose the document contains this sequence of lines:

Line0, Line1, Line2

Suppose Line1 has a space at the start. That means Line0 is continued on Line1 and unfolding will merge the two lines into one. After unfolding, the document has this:

Line0', Line2

where Line0' contains the data from Line0 plus Line1.

**pred** unfold (doc, doc': Document, line: doc.lines) {  
 // precondition: “line” has a next line (i.e., it's not the last line)  
 **some** line[Line]   
 // precondition: the next line starts with a space  
 **some** line[Line].spaceAtStart  
 **let** line1 = line.Line, line2 = line[Line], line3 = doc.lines[line2] {  
 // Merge line1 and line2. Set line3 to follow line1.  
 // Add the content of line2 to line1. Delete line2 -> line3.   
 // Unless ... no line follows line2. Then the unfold  
 // results in no line following line1. Remove line1 -> line2.  
 **some** doc.lines[line2] => doc'.lines = doc.lines ++ (line1 -> line3) - (line2 -> line3)  
 **else** doc'.lines = doc.lines - (line1 -> line2)  
 doc'.content = doc.content ++ (line1 -> (doc.content[line1] + doc.content[line2]))  
 doc'.firstLine = doc.firstLine  
 }  
}

Initialize the document with some lines. Give each line unique data. Make some lines that will require unfolding.

**pred** init (doc: Document) {  
 **some** doc.lines  
 **let** allLines = doc.lines[Line] + doc.lines.Line {  
 **all** line: allLines | **let** data = doc.content[line] | **one** data **and**   
 **all** otherLine: allLines - line | doc.content[otherLine] != data  
 **some** allLines.spaceAtStart  
 }  
}

At each step of the execution trace execute the unfold operation, if there is at least one line that needs unfolding. Otherwise, do nothing to the document (i.e., do a skip).

**fact** traces {  
 init [first]  
 **all** doc: Document - last | **let** doc' = doc.next {  
 // If spaces exist in the document, do an unfold operation.  
 // Otherwise, do nothing (i.e., skip).  
 **let** allLines = doc.lines[Line] + doc.lines.Line {  
 **some** allLines.spaceAtStart => not skip [doc, doc']  
 **some** line: doc.lines | unfold [doc, doc', line] **or** skip [doc, doc']  
 }  
 }  
}

"skip" means no change to the document, i.e., the next state of the document is the same as the last state.

**pred** skip (doc, doc': Document) {  
 doc'.lines = doc.lines  
 doc'.content = doc.content  
 doc'.firstLine = doc.firstLine  
}

Assert: The final state of the Document has no lines folded.

**assert** no\_lines\_are\_folded {  
 **let** allLines = last.lines[Line] + last.lines.Line |  
 **no** allLines.spaceAtStart  
}  
**check** no\_lines\_are\_folded **for** 6

The Alloy Analyzer finds no counterexamples. Yea!

## Implementation

In implementing unfolding I adhered as closely as possible to the model. I used XSLT as the implementation language.

Whereas the Alloy model has a set of lines representing all possible iCalendar files, the implementation can operate only on specific iCalendar files, so I opened a text editor and typed in this sample iCalendar file (sample.ics):

BEGIN:VCALENDAR  
VERSION:2.0  
CALSCALE:GREGORIAN  
BEGIN:VEVENT  
DESCRIPTION;ALTREP="cid:part1.0001@example.org":The Fall'98 Wild  
 Wizards Conference - - Las Vegas\, NV\, USA  
END:VEVENT  
END:VCALENDAR

My implementation first reads in sample.ics as a long text string and then breaks up the text along newlines. The variable, $Document, holds the sequence of strings (lines).

<xsl:variable name="raw-text" select="unparsed-text('sample.ics', 'utf-8')"/>  
<xsl:variable name="Document" select="for $i in tokenize($raw-text, '\r\n', 's') return $i"/>

The unfold operation is implemented as a function. The function has two parameters which must be given two consecutive lines from the Document. It returns one line if unfolding was done, or two lines if no unfolding was done. The model has two preconditions – line2 must exist and it must have a space. The preconditions are implemented with a conditional choice. If the preconditions are satisfied, the function returns a single line that is the concatenation of the two parameters. Otherwise, the function returns both lines, unaltered.

<xsl:function name="f:unfold" as="xs:string\*">  
 <xsl:param name="line1" as="xs:string" />  
 <xsl:param name="line2" as="xs:string?" />  
   
 <xsl:choose>  
 <xsl:when test="$line2 and starts-with($line2, ' ')">  
 <xsl:sequence select="concat($line1, $line2)" />  
 </xsl:when>  
 <xsl:otherwise>  
 <xsl:sequence select="($line1, $line2)" />  
 </xsl:otherwise>  
 </xsl:choose>  
</xsl:function>

The execution trace in the model iterates over each state of the Document, invoking the unfold operation when there are lines with spaces, and a skip otherwise. The implementation is not as straightforward. Recursion is used to mimic the states of the Document. Only one pass is made over the lines. An index variable points to the current line being processed. The value returned by the unfold operation is checked: if two values are returned, then no unfolding was done, so recurse and increment index; otherwise one value was returned (and the value is the concatenation of the two lines), so recurse but do not increment index because the current line must be compared against the new next line. In the model the Document’s next state is simply expressed. The Document’s next state is not so easily expressed in the implementation: the Document must be disassembled, the appropriate parts identified, and then reassembled. The assembled parts are then passed into the recursive call.

<xsl:function name="f:trace" as="xs:string\*">  
 <!-- Document contains a sequence of lines -->  
 <xsl:param name="document" as="xs:string\*" />  
 <!-- The line to process next -->  
 <xsl:param name="index" as="xs:integer" />   
   
 <xsl:choose>  
 <!-- At the last line? -->  
 <xsl:when test="$index ge count($document)">   
 <!-- Then return the unfolded document -->  
 <xsl:sequence select="$document" />   
 </xsl:when>  
 <!-- Not at the last line -->  
 <xsl:otherwise>   
 <!-- Invoke the unfold operation, passing it the current line and next line -->  
 <xsl:variable name="result"   
 select="f:unfold($document[$index],$document[$index+1])" />  
 <xsl:choose>  
 <!-- If the unfold operation returns a single line, then it unfolded the two lines.  
 Concatenate: the lines up to index, the result returned from the unfold  
 operation, and the lines after index+1. Recurse, assign $document the concatenated  
 lines, assign $index the current value of $index. -->  
 <xsl:when test="count($result) eq 1">  
 <xsl:sequence select="f:trace(($document[position() lt $index], $result,   
 $document[position() gt $index+1]), $index)" />  
 </xsl:when>  
 <!-- If the unfold operation returns two lines, then no unfolded took place.  
 Concatenate: the lines up to index, the two lines, and the lines after   
 index+1. Recurse, assign $document the concatenated  
 lines, assign $index the result of incrementing $index. -->  
 <xsl:otherwise>  
 <xsl:sequence select="f:trace(($document[position() lt $index], $result,   
 $document[position() gt $index+1]), $index+1)" />  
 </xsl:otherwise>  
 </xsl:choose>  
 </xsl:otherwise>  
 </xsl:choose>  
</xsl:function>

Lastly, the execution trace is invoked, which returns a sequence of unfolded lines. This is then output with a newline inserted between each line.

<xsl:template match="/">  
 <xsl:variable name="results" select="f:trace($Document, 1)" />  
 <xsl:for-each select="$results">  
 <xsl:value-of select="."/>  
 <!-- Insert a newline after each line -->   
<xsl:text>  
</xsl:text>  
 </xsl:for-each>  
</xsl:template>

Whereas the model can be analyzed by the Alloy Analyzer, the implementation cannot be analyzed by machine. Instead, a bunch of tests must be created and run.

## Lessons Learned

The model gave clarity to the key ingredients in the unfold problem, without getting bogged down in coding irrelevancies such as, should I use recursion or should I use a for loop? how to represent the Document? how to disassemble and then reassemble the Document? And so on.

A machine – the Alloy Analyzer – checked the model. I asked it to check that Document would always end up in a completely unfolded state. It said yes. That is awesome. That kind of machine-analysis cannot be done on code.