Systematic validation of localisation across all languages

Martin Ørsted
Microsoft Ireland
Martin.Orsted@microsoft.com

Abstract

As software companies increase the number of markets and languages that they release their products in, it may become necessary to change the localisation process for these products. Quality assurance (QA) is often viewed as an area where processes could be streamlined through automation and one method for doing this would be through the design of a localisation verification system that can validate single resources across languages as well as check for generic issues across multiple resources and languages. This article outlines a graduated approach to systemically capture and fix issues when a product is being localised into an increasing number of languages. By examining multiple languages, patterns can be identified that enable the identification of inconsistencies and issues that, with a single language approach, would have been very costly and difficult to unearth.

Keywords: *Localisation, Resources, Verification, Systematic, Multiple languages, Controlled language*

Introduction

The best place to address localisation issues is upstream. Much can be done here; the use of newer programming languages with more built-in error checking, the use of pseudo localisation[1] upstream, educating developers, the use of controlled English and source reuse systems can all help. There are however many reasons why the above options will never be implemented perfectly; deadlines, tradeoffs, the inadequacy of the development languages used, and so on. For these reasons systematic validation can improve localisation and noticeably drive down costs for multi-language releases so that the more languages produced the better the return.

In most traditional localisation efforts languages are treated rather independently, with little or no ability to leverage the testing performed for one language on another. The most common forms of leveraging are highly manual or risk based[2] or a combination of both. One way of leveraging is, for example, to not run low priority test cases on certain languages; another is manual regression of bugs found in one language against others. The use of pseudo localisation is also gaining broad acceptance in the industry and serves many needs. Using pseudo localisation with machine generated pseudo localised strings will allow for a fast check of the localisability of resources and can in general find most types of localisability errors up front. In this context pseudo localisation is often used to postpone the real localisation effort until RTM or RTW (Release to manufacturing or web) or at least shorten the parallel effort, which reduces resource churn and drives down localisation cost. Used in this way it also saves on development costs as the faster an issue is found the cheaper it is to fix it.

There are several goals behind the systemic validation of localisation across all languages. This article uses practical work that has been carried out in Microsoft over the years to map out how a methodology can be built around using systemic validation that can achieve higher savings and better turnaround times than the aforementioned approaches can deliver. The article will start by looking at the single resource approach, where Microsoft's rules based approach is explained, and over the course of this section it will show the kind of issues that one can systematically fix. It will then generalise the approach to a wider pool of resources. We will look at other methods for bug avoidance, and finish up by analysing how testing can systemically be reduced while quality is maintained, or improved, through the introduction of the outlined methods. In this way we will also look at how the traditional linear dependency between the cost of the test effort and the number of languages localised can be broken.

[1] Pseudo localisation is localising strings by replacing the typically US characters with characters from other code pages, and adding tagging before and after the string. Open could for example become become \?p??$@#. Typically the pseudo localisation process is fully automated so it is fast and cheap.

[2] Risk based through the use of orthogonal arrays for example.

The single resource

There can be many reasons why the localisation of a string can cause a bug, be it user interface or functional. In the functional space bugs can be caused by:

- Over-localisation: The string should not have been translated.
- Buffer limitation: The translation of the resource should not be more than a given amount of characters, generally referred to as a string length limitation.
- Illegal characters: Certain characters may not be allowed in the string
- Dependency: Two resources may have to be translated as one, in effect one resource is dependent on the other, references the other.
- Backward compatibility: This is a special case of dependency, basically where changing a string from one version to another could cause a loss of backward compatibility.
- Uniqueness: The string belongs to a group of strings that all have to have unique names (translations), for example, a list of commands.
- Placeholder over-localised: Some localisable strings have placeholders in them. If the placeholder gets localised the program cannot drop the information into the placeholder and display it.
- Required string decoration: Some strings may have control characters in the beginning or end of the string that should not be localised

There are other more special cases, but the above list captures the majority. In many instances strings that can break because of any of the above causes could have been bullet-proofed by the developer, but it is not always the case.

Below are a few examples of strings that would fall into these categories:

Rule	US string	Example loc	Issue description
Over-localisation	Common Files		Might refer to a registry string. Rather than localising the string the program will look up the localised name in the registry
Placeholder	The file %1 could not be opened because %2		%1 and %2 are placeholders
Decoration	\n\nOpen\n\n		\n is a new-line character, sometimes used in command line applications
Placeholder	The file %s was last opened on %d %d	On %d%d the file %s was last opened	%s and %d are positional placeholders, their position has to be maintained, changing them as shown will cause an intermittent memory protection fault

In Microsoft the original approach we had to systematically fix issues once we identified them was LocVer, short for Localisation Verification. For us, LocVer is still an essential part of our strategy. With LocVer, we can create rules to describe the limitations for a given string, and we can then run the rule engine against all languages. By doing so, we can ensure that the issue, once found, is validated and if needed fixed for all languages.

3 LocVer is Microsoft patented and patent pending

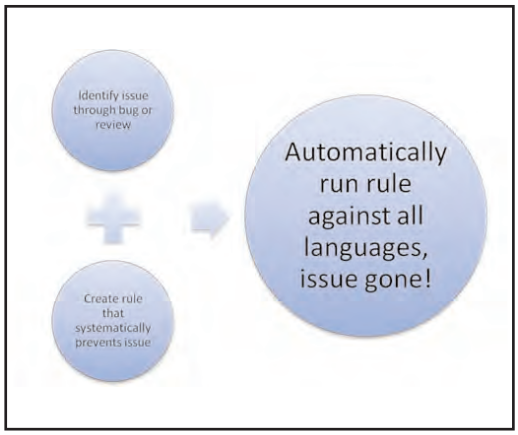

FIGURE 1: A GRAPHICAL REPRESENTATION OF LOCVER

Figure 1 shows a graphical representation of the idea. So as a hypothetical, practical example, the string **Current Accounts** could be used in Microsoft Excel, perhaps it becomes part of a Pivot table, and let us assume that we have identified, through trial and error, that the string cannot contain more than 30 characters. At this point in time an engineer would often have the choice of either transferring the bug to the developer to increase the buffer or accepting the limitation. We choose to accept the limitation and we create a LocVer rule: **MaxLength=30** (meaning that the translation is not allowed to contain more than 30 characters). Since we associate instructions with each string, the rule becomes an instruction for that particular resource ID.

We have a master repository for instructions across all languages, and that is where our new instruction will be added. The system is designed so that the localisation vendors frequently receive the latest instructions and has been designed to run this type of validation at each handoff, so in effect given a certain lapse time the rule will have migrated across to all languages, and for any language where the rule has been broken an error will be returned and logged, so that the issue can be resolved.

LocVer has been in use in Microsoft for many years now and has developed from supporting simple rules like the one above to more complex scripting rules. As we progressed with this we realised that there was a need for added functionality, such as the ability to conditionally apply a LocVer rule, perhaps the rule should only apply to a subset of languages, or maybe a subset of languages should be excluded. You can have strings that accept ANSI 1252 characters, but where Cyrillic characters may cause an issue, and hence you may lock translation for those. Or you may have a program where some advanced functionality is available for a few main languages (speech recognition for example), but since it is not available for other languages the items should not be localised.
There is a cost involved in this per resource based instruction approach. Whenever a resource changes, the associated instruction will have to be updated, so the adding of rules and their maintenance can become a serious effort. Measuring the return on investment can also be a little difficult, in effect how do you account for the bugs prevented?

Still, in many instances the individual instructions are often the only viable way of dealing with per string limitations. However this is not always the case and that is what the next section deals with.

Fixing systematically across a wider pool of string resources

The previous section dealt with a per resource approach to the systematic validation of resources. It works, it is proven and we use it a lot. However, the cost involved means that we have had to consider whether we could further develop the approach in such a way that all the benefits of the above system can be retained but without the management overhead of dealing with the individual resources.

There are several different approaches that can be considered to reduce the management overhead. At Microsoft we have, in practical terms, at least three concurrent systems in place, each serving different needs. One way of approaching it is to see if we can create generic rules. Where this is possible we can then remove many specific rules and rely on a few generic rules instead. This turns out to be very applicable for certain placeholders. If for example %1, %2 and %3 always denote placeholders, then we can remove the specific LocVer placeholder rules from the individual resources and create a generic rule that stipulates that %1, %2 and %3 are always placeholders and that the translation has to mirror the source in their usage. This is an approach that we use frequently to avoid functional issues.

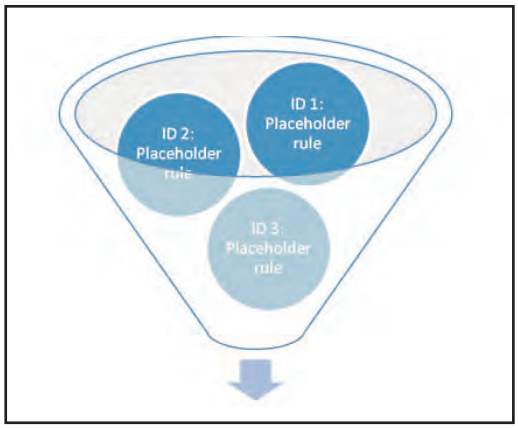

FIGURE 2: A GENERIC PLACEHOLDER RULE WITH NO ASSOCIATED ID

In terms of ensuring that certain keywords and copyright text are correct it is possible to do something quite similar. As a case in point, in the old days we used placeholder rules to ensure that certain legal text was kept across languages, but this is inappropriate for several reasons, not least of which is the maintenance associated with this method. In parallel with the generic rules we therefore keep a list of SQL queries that we can run to ensure that copyright text and application names are treated consistently.

This approach can be used, for example, where a product is developed with a code name up to a very late stage when the official product release name is decided upon. So, for example, InfoPath was referred to as XDocs during the development phase, and it was only very late in the process that it became InfoPath. In this case a simple SQL query could be run against all languages to identify places where localisers had "forgotten" to change the name.

We have built up a list of checks, either single language checks or relative to the US English source that we can run in these situations. For example, in the last release of Microsoft Office we might have run checks to find strings where the US source had "Copyright" in the text but the localised text didn't, or the US version had "2007" in the source but the translation did not contain it, or the US version had "Microsoft Word" in the source but the translation didn't.

There are many reasons why a more explicit per resource rule-based approach is not acceptable here. Firstly, it would involve far too many rules and drive up cost. Secondly, there is an advantage to having clear separation between the functional quality assurance and the legal/linguistic assurance. The functional quality assurance process is ongoing and needs a continuous focus. In general you expect very few false positives, in effect rules misfiring. As opposed to those for linguistic/legal material the rules can misfire frequently, and the linguistic quality issues are not as time critical as the functional ones; the functional quality has to be high continuously. The reason for this will be made apparent below.

A functional issue may block the testing of an area, and consequently the fixing of the functional issue may in turn uncover more issues. As opposed to that a linguistic issue may often be benign, not an issue at all. The string **"Microsoft Office Word could not open the file %1"** could, for example, in some languages be translated to the equivalent of **"The file %1 could not be opened"** as a space saving measure, and that would in many instances be quite OK. To complicate matters more, we would allow the use of the Cyrillic "i" instead of the Latin "i" for Cyrillic languages, hence the word **Microsoft** would not even match up for Cyrillic languages in a comparison between the language and US.

In terms of the legal/linguistic searches we find therefore that there is a trade off point, after which it is not worthwhile. We would tend to run our queries a couple of times during production and pay the localisation vendor to review the results, calling out the ones that need fixing and getting them fixed. Geopolitical issues can be dealt with in a similar way. We maintain lists of words or phrases that are geopolitical on a per language basis, and we can run them through the same system that we use for legal quality.

Adding the language dimension

Over the last 10 years the amount of languages we localise into at Microsoft has seen a dramatic increase. When I started as a localiser in 1996 we probably localised into around 15 languages, now we are getting closer to 100, if not exceeding this number. This pattern seems to be repeating itself within the industry as a whole. The way we approach localisation changes with the addition of more and more languages. Approaches that would previously have been too costly start to become viable. Likewise, certain approaches become possible that before would have been impossible. That is the topic of this section. We also, conveniently, enter the newer and most exciting or promising areas of localisation innovation here. Whereas there are innumerable examples relat-

ing to the last two sections, here there are fewer examples and, of the examples that there are, some may be rather theoretical.

In the previous section I touched upon running queries against a target language and source language (US), to find product names and copyright issues. With the addition of the language dimension smarter queries can be run that look across multiple languages to find patterns. If nine out of ten languages turn out to start or end with a certain sequence of characters, or if for example the word **Microsoft** appears in nine out of ten, then there is good reason to assume that it should be in the last language as well, it becomes a pattern that triggers an exception for evaluation. The return on investment (ROI) on creating various different rules obviously goes up with more languages added, so this is a space that is open for creativity.

One thing you can systematically look for is true repeated strings. Quite often, just because two US strings are identical one cannot assume they carry the same meaning, **Open** can be the imperative, as in the command **Open the door**. It can also be the infinitive, to **Open**, and the two are translated differently for most languages. But if the strings are the same for US and nine out of ten localised languages, then the deviation for the tenth language is probably an error. Going further, one could for all languages after, for example, the tenth language simply remove identified repeats and only have them translated once per language.

Another effort that we have invested in is tweaking our pseudo localisation engine, so that it understands and adheres to our LocVer rules. That means we only ever find the same issue once, since the pseudo localised strings won't break the rules we have already added. Pseudo localisation on various different languages is therefore a key part of our strategy. The question becomes "what further testing needs to be carried out on fully localised languages?" Figuring this out involves analysing exactly what kind of testing needs to be performed on the actual languages themselves, and that means analysing the localisation model.

The localisation model in this context is the various processes that are applied to get from the US English source files to the localised files. Each process needs to be analysed to figure out what error sources the process can introduce and what error sources are systematically prevented. For each error source identified that requires testing to revisit the various languages, the challenge is to identify a solution that can systematically fix the root cause of the issue so that the need to test the various languages is kept to a minimum or eliminated in certain instances. DAL (Dynamic Auto Layout of dialogs) for example may, implemented correctly, mean that it won't be necessary to review each localised dialog, but rather a subset, or just the pseudo dialog depending on confidence levels. These confidence levels are subjective and based on experience.

Similarly, we have introduced a systematic way of assigning hotkeys, so that the assignment of hotkeys per language is really a matter of running a set of scripts at a chosen moment. This is accomplished by building up a list of all resources with hotkeys and where they appear. Building up that list is complicated, it is partially populated through the use of automated trawler tools that identify which resources belong to which dialogs, but it can also be populated or improved upon through manual entries. So the process is a bit costly. With the lists populated we can then run a tool on a per language basis that knows which characters can be assigned hotkeys per language, and that can resolve hotkey conflicts including dealing with resources that appear in multiple dialogs or menus.

Other concerns that need to be addressed are the behaviour of the product with various code pages. So one thing that is possible is to group languages under ANSI code pages and test a representative from each code page exhaustively. Again, getting full Unicode from scratch would be an advantage of course, but failing that this approach can drive down cost.

With all of the above implemented, or parts of it, one can evaluate the approach to testing. Engineering are in a position to guarantee that testing only ever needs to find an issue once, and engineering can guarantee that it will systematically be fixed across all languages with no further test need for verification. That in turn can facilitate moving away from very specific test cases to higher level Test Design Specifications. This is helped by the fact that the testers, assuming you use the same testers for all languages, gradually build up a better understanding of the tested product and the type of localisation bugs that appear. So rather than executing very specific test cases step by step, overall quality can be improved at a reduced cost by having the testers test features with some high level guidelines that lead them through the features but are still much more

abstract than specific test cases. The result will be that the tester for each language will go through a feature based on his or her high level of understanding of this feature, and since the tester is not following a strict script we can count on a degree of variation or randomness to be introduced in the ways the various languages are tested. The introduced randomness that this brings adds value to the process, because through the testing of various languages, pseudo or real, the randomness introduced will ensure better overall test coverage as compared to strictly exercising the same test paths per language.

We saw an example of this in a recent Office release cycle. We had a test case that stipulated the comparison of two files, and included the two files as part of the test case. One tester chose to compare two files that were already on disk rather than take the two supplied for this test case and as a result uncovered a bug that for several languages had not been found.
All of the above efforts should mean that the test and engineering cost per language can be reduced. In an ideal world the cost of adding an extra language should get close to the cost of the pure translation; realistically in our case we have not achieved that, but we have definitely seen dramatic cost reductions. It is difficult to give exact savings numbers, due to a number of factors such as the difficulty in calculating the savings of a bug that has been avoided and also because we have introduced these efficiencies gradually, and in some cases are still working on fully introducing them. But to give an idea of the importance of this, the group that I work in is requested to make serious savings version on version, and this is one of our favourite hunting areas for those savings.

Controlled English and Machine Translation
Another benefit of localising into many languages is that more structured approaches within the area of controlled English and Machine Translation (MT) become feasible. So at some stage it makes sense to use controlled English or elements of controlled English, starting with simple checks on sentence length and verbs in passive tense and moving on from there.

Machine translation is trickier. There is no standard emerging in the MT space to automate translation in an intelligent way. Also, most MT engines go from English to another language, but much more can be gained with an effort that translates between close language pairs, for example Iberian Portuguese and Brazilian Portuguese, or Norwegian Bokmål and Norwegian Nynorsk. The localisation verification models can ensure that rules are not broken and cost can be reduced, although an initial investment is needed. As an alternative to full MT, automated transliteration can be considered from some languages for language pairs that are closely related.

Conclusion
There is no substitute for a well engineered product in the first place. Any bullet-proofing that can be done within the code is of course preferred. In the real world there will however always be limitations to the upstream efforts, and that is what we are looking at here.

Figure 3: Evolution and return on investment of various approaches

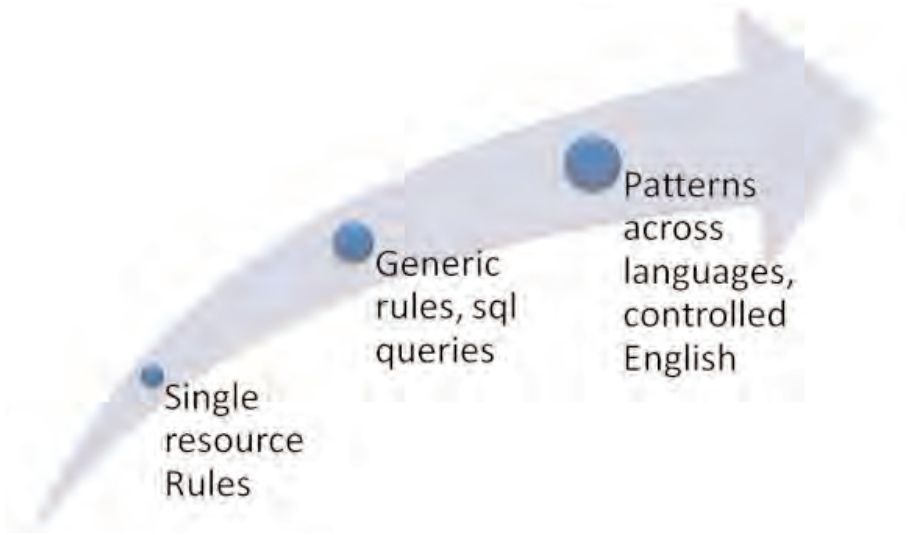

Figure 3 illustrates the evolution and the return on investment of the various approaches. The further to the right the greater the impact, but this doesn't mean that single resource rules are not valuable, just that they are rather costly in comparison to the other approaches.

The above sections have outlined a graduated approach to systemically capture and fix issues when a product is being localised into an increasing number of languages. We began by looking at approaches to individual strings. This is often a necessary approach in functional bug fixing and prevention, and on the positive side it means that we only ever have to catch a specific issue once, and we can then systematically ensure that, should the issue occur in other languages, an error will be raised and we can deal with the issue manually or automatically. The downside to this approach is that it necessitates either an inspection of all resources, where certain kinds of issues cannot be found, or that the issue is identified as a bug at some stage in a language. The other downside is the actual cost of running a system like this; depending on the thoroughness applied the cost can be quite severe.

A system where the rules are generic is therefore preferred, but will never be able to cover everything. The advantages of the generic rules are that they look for identifiable patterns and automatically apply when a pattern is identified. Therefore, new strings that conform to the same patterns, for example %1 as a placeholder, are automatically covered as soon as they are added. String changes, the addition or removal of placeholders, will automatically be covered, and the management overhead is dramatically reduced in comparison to the single resource rules. Similar benefits of scale can be achieved in the legal and linguistic space through the use of SQL type queries.

Finally, with the addition of multiple languages pattern recognition across languages becomes interesting. Certain types of errors become much easier to detect, and things like controlled English make it possible to ensure a higher end localisation quality (because the localisable text is less ambiguous).

It is therefore possible to use the fact that a product is localised into many languages to systematically deal with some issues, to apply learning across the languages that can help raise the overall quality of the product, and to drive down the cost of testing and bug fixing. This approach has the potential to break, what is often, otherwise, a linear dependency between the number of languages you localise into and the total cost of bug fixing and testing.