Extreme Maintenance: Transforming Delphi into C#

John Brant
brant@refactoryworkers.com

Don Roberts
University of Evansville
roberts@evansville.edu

Bill Plendl
ProfitStars®
A Jack Henry Company
bplendl@profitstars.com

Jeff Prince
ProfitStars®
A Jack Henry Company
wjprince@profitstars.com

***Abstract*—Sometimes projects need to switch implementation languages. Rather than stopping software development and rewriting the project from scratch, transformation rules can map from the original language to the new language. These rules can be developed in parallel to standard software development, allowing the project to be cut over without any loss of development time, once the rules are complete. This paper presents a migration project that used transformation rules to successfully convert 1.5 MLOC of Delphi to C# in 18 months while allowing existing Delphi development to continue.**

I. Why Change Languages

Software maintenance is generally about making incremental changes to a code base over a long period of time. However, there are sometimes large, system-wide changes that need to be made to modernize the code (e.g. changing frameworks). The most extreme of these types of changes is completely changing the implementation language.

It is generally recognized that rewriting a software system from scratch is a bad idea [1]. However, there are still valid reasons for switching languages. These include difficulty in hiring qualified programmers, limited availability of third-party libraries, lack of vendor support for the language tools, limited or delayed support for new technologies, and enterprise integration after an acquisition.

For all of these reasons, ProfitStars®, a division of Jack Henry & Associates, decided to migrate their core software assets from Delphi to C#. They were increasingly finding it difficult to hire and retain new programmers that were experienced in Delphi programming. Third party libraries that their products relied on were becoming unsupported and libraries for new functionality were not available. But the most compelling reason for their switch was ProfitStars' desire to utilize the latest .NET technologies with their legacy applications. Another contributing factor for the migration was that they were acquired by Jack Henry & Associates, which primarily used C# in their programs. Integrating the PROFITstar Asset/Liability Management (ALM) software with the software products of the larger organization was critical and difficult to do under Delphi.

ProfitStars' core ALM software assets are three major software packages, several smaller software tools, and corresponding automated test suites comprising approximately 1.5 MLOC of Delphi. By using the techniques and tools described in this paper, we were able to translate their code base into 1.5 MLOC of C# in 18 months using only four developers, ensure that all of their automated tests continued to pass, and pass human QA testing, all with minimal impact on their day-to-day software development process. During this time, a major release occurred, new features were added and bugs were fixed, all in the original Delphi code, which was reflected in the translated C# code. On the cut over day, the translated C# code was up-to-date with the latest Delphi code.

II. Translation Processes

Several approaches have been tried to reimplement large software projects.

A. Redesigning Rewrite

The most common approach is the rewrite. In this, software development on the original system halts, or at least is significantly reduced, while the team manually reimplements the system in the new language. This reimplementation may be a complete redesign or a manual conversion of the existing source. There are several problems with this approach. First, during the rewrite, few features or fixes are added to the original code since the developers are busy implementing the new system. Any fix that is added must be explicitly managed to make sure it is included in the new system. In the best case, during the rewrite, customers will see no changes. This causes the company to lose ground to its competitors. The second problem is that redeveloping the software is nearly as error-prone as developing it the first time. New bugs will be introduced. What exacerbates this is the tendency for groups to redesign as they reimplement, attempting to simplify complex code. Often, the original code was complex because it needed to handle corner cases that have since been forgotten. These cases often get lost in the reimplementation. As a result of these problems, many rewrite attempts completely fail.

ProfitStars considered this approach, but rejected it for two reasons. First, the estimate for the time required to rewrite the project was measured in years rather than months. Second, they did not think that it was feasible to keep two separate code bases in sync for that period of time.

B. One-Shot Transformation Tool

Another approach is to use a tool that converts the program's syntax and some of the more common types to the target language. Once this initial conversion is made, the developers manually make the translated code run. Like the rewrite approach above, all new development stops until the program gets back into a working state. While this approach is generally faster than the manual reimplementation, it has a major problem in that there is no feedback to make the initial conversion better. When there are problems with the conversion, either the developers have to manually fix them every place they occur, or they can rerun the conversion with the update and lose all of the manual edits that have already occurred. Often, these global problems are not uncovered early enough so the approach is to manually fix all the occurrences.

C. *Rule-Based Transformation*

Our approach for migrating is to build a set of transformation rules that convert the code [2]. This approach is not new [3] and is supported by several tools [4][5][6]. While similar to the one-shot transformation tool approach, the important distinction is that instead of editing the transformed code directly, we edit the rules that transform the code. Any edits made directly to the code will be lost when the code is retransformed, whereas any changes made to the transformation rules will be applied globally. Our tools [7] have been used along with similar techniques to perform massive transformations such as replacing an application data layer on a large project [8]. While creating a rule may take longer than editing the code directly, it has some advantages. The biggest advantage is that development of the original code base can continue in parallel with the construction of the transformation rule set. Because the source is never directly edited, the rules can be replayed at any time to translate the latest code. For the PROFITstar ALM project, the code was translated on a daily basis so new code written one day in Delphi was available the next day in C#. Another big advantage with the rule-based approach is that the rules are applied globally over the entire project. A rule that fixes one piece of code will likely fix several other locations that were previously unknown.

While a redesigning rewrite could potentially have the greatest reward by having a better designed, more maintainable system; it also has the largest risk. There is no guarantee that the new code will be substantially better than the previous to make up for the much greater implementation costs. Furthermore, since the rewrite will take longer than other approaches, it also has the greatest risk of losing customers since their bugs and new feature requests are not being addressed. The transformation rule set approach is much better at managing these risks. Not only is it much quicker to convert the existing code instead of redeveloping the code, it has minimal impact on the existing development process of fixing bugs and adding new features.

One drawback of the transformation tool and rule set approaches is that the generated code may not adhere to the standards and practices of the target language. For example, one major difference when converting from Delphi to C# is naming conventions. In Delphi, class names are normally prefixed with a "T" and instance variable names are prefixed with an "F". Neither prefix is common in C#. Another major difference for a Delphi to C# conversion is temporary variable declarations. In Delphi the temporary variable is declared separately from its first use. However, in C# it is common for the variable declaration to appear on the same line as the first assignment in a method. As a result, after the code has been translated, it looks more like Delphi code than "standard" C# code. Many of these standards and practices can be applied after the migration is complete by using a refactoring tool like ReSharper[1] or by writing custom refactoring transformation rules. Keeping these refactoring transformation rules separate from the migration transformation rules helps simplify the migration rules since we do not need to think about possible interaction between the two sets of rules.

```
Inc(`a`, `b`) ⟼ `a` += `b`
```

Fig. 1. Pattern to translate Delphi Inc expression into C#

Another possible shortcoming of the rule approach is the "compatibility layer" that you often have to build. If a library in the old language is substantially different from the new library, often it is easier to create a new component in the new language that behaves like the old one. If we were doing a strict rewrite, we would not need these "extra" components, so there is a bit of cruft that appears in the design due to this approach. However, this cruft can be removed by refactoring after the migration is finished.

III. The Tool

We implemented all of the transformation rules for the project using SmaCC and the SmaCC Transformation Toolkit. SmaCC is a parser generator for Smalltalk. It can generate abstract syntax trees (ASTs) along with a corresponding visitor that will traverse the AST. ASTs generated using SmaCC have pattern matching and rewriting support.

A. *Rewrite Patterns*

Pattern matches are special code fragments in an augmented form of the original language that parse into pattern ASTs. They can contain special pattern variables that can match subtrees in that position. For most languages, a pattern variable is anything delimited by the back-quote character (`) since that character isn't commonly used.

Figure 1 shows a search and replace pattern for translating the Delphi Inc statement into a C# += statement. It has two pattern variables: ``a`` and ``b``. These pattern variables allow us to match both Inc(var, 3) with ``a`` = var and ``b`` = 3 and Inc(obj.var, 3 + 4) with ``a`` = obj.var and ``b`` = 3 + 4. If the same pattern variable is used multiple times, then it must match the same expression in each position.

Unlike the search patterns which must parse into an AST, the replacement patterns can be arbitrary text. This text directly replaces the source of the matched AST node. Whenever

[1]http://www.jetbrains.com/resharper/

```
`a/String`[`i`] ⟼ `a`[(`i`)-1]
```

Fig. 2. Pattern to translate one-based indexing of strings into zero-based indexing

a pattern variable occurs in the replacement expression, the transformed source of whatever it matches is copied into the replacement. While these replacement patterns look similar to those used by other tools [4][7][5][6], they are quite different. In the other tools the replacements transform ASTs to other ASTs which are then pretty-printed. In SmaCC, the source code associated with the AST is directly transformed via string edits. By using text for the replacement, the replacement expressions are not tied to any particular language. At the beginning of the project we used this functionality to generate reports of problem areas in their code by transforming patterns of problem code directly into error messages.

For the project, we augmented the patterns provided by SmaCC with type information. This allowed us to create pattern expressions that would only match AST nodes of a certain type. For example, Figure 2 shows a pattern for translating the one-based string indexing that Delphi provides to zero-based indexing in C#. The search pattern variable, `` `a` ``, has been extended with the type `String` so it will only match AST nodes that have the String type in Delphi. This rule may insert unnecessary parentheses around the `` `i` `` pattern node. However, these were easily removed with C# to C# pattern rewrites that were applied at the end of the project. Furthermore, these types of C# pattern rewrites can remove the "`+ 1 - 1`" that may occur from this rewrite when `` `i` `` contains "`+ 1`".

B. The SmaCC Transformation Toolkit

The SmaCC Transformation Toolkit is our program that allows us to build a set of transformations rules using the parsers generated from SmaCC. Each transformation rule has two parts: the search expression and the transformation that is applied when the search expression matches. Search expressions can either be SmaCC patterns or Smalltalk expressions. The Smalltalk expressions are arbitrary pieces of code that return true when they match a node. Certain types of matches are easier and faster if we can explicitly test the AST node using a Smalltalk expression rather than a pattern. For example, when converting Delphi to C# there are many syntactical changes that are performed solely based on the class of AST node (e.g., if-then-else node). While we could write patterns for these, it is easier and faster to simply write a Smalltalk expression that checks the class of the node.

Similar to the search expressions, replacement expressions can either be SmaCC replacement patterns or Smalltalk expressions. The Smalltalk expressions allow arbitrary Smalltalk code, but they generally fall into two different types of expressions. The first type is simple edit expressions such as inserting, deleting, or moving text. For example, if we are transforming an assignment expression in Delphi to a C# assignment expression, we need to replace the "`:=`" with an "=". We can do that with "`self replace: match assignment with: '='`". The other common type of Smalltalk expression is setup code. This setup code can define values that are used throughout the transformation process. For example, we have an expression that loads a spreadsheet that defines how to map Delphi types and methods into C# types and methods. This spreadsheet can be shared and edited by people not familiar with SmaCC's patterns or Smalltalk.

Using Smalltalk expressions gave us several advantages over building a separate transformation language. First, we already knew the language. Second, we could use the existing development tools. We didn't need to write custom compilers, debuggers, and inspectors. Third, it allowed the full environment to be used. This allowed us to use existing Smalltalk code that dealt with files, images, etc. without need to define them in a special transformation language.

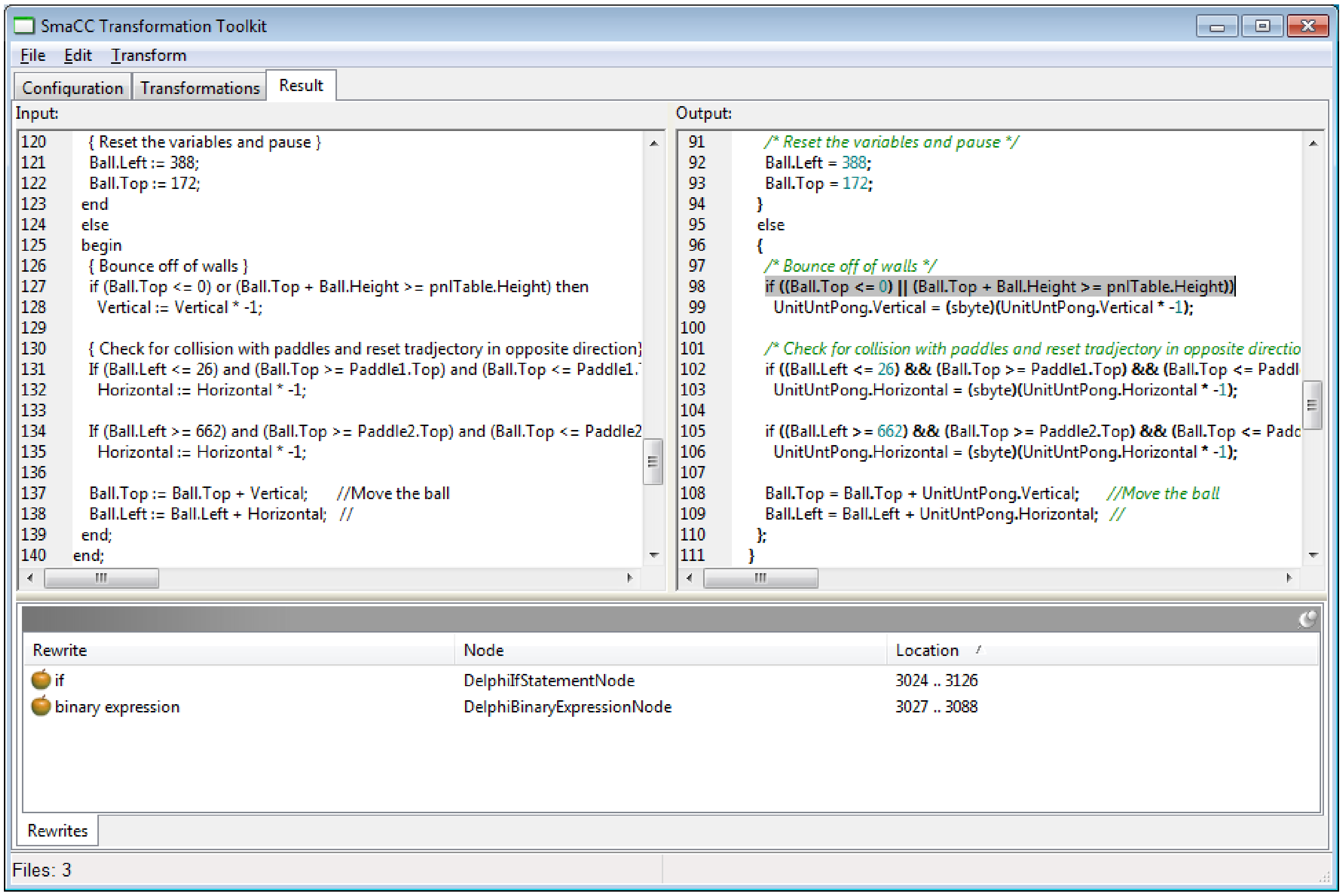
With hundreds to thousands of transformation rules, it can become difficult to figure out which rule is affecting a certain piece of code. The SmaCC Transformation Toolkit has support for showing which transformations affect a piece of code. The user can select text from the input or output and see which rules affected the selected text. Furthermore, selecting the rule will underline the actual edits in the input and output panes. Figure 3 shows the Delphi input and the C# output of a simple pong game. In the output, an "`if`" statement has been highlighted. Two rewrites affected this output. The first rewrite is the "`if`" rewrite which added the parentheses around the expression and deleted "`then`", and the second rewrite is the "binary expression" rewrite that converted the Delphi "`or`" into a C# "`||`".

In addition to transforming the Delphi code, we also wrote transformation rules to transform the Delphi form files (.DFM files) into C# designer files. The DFM files were not Delphi code, but a list of properties for each GUI component in an interface. This was effectively a different language that we were transforming into C#.

To help with scaling a migration project, the SmaCC Transformation Toolkit can distribute the work of transforming the files across several machines and/or processes. Since the goal was to build a set of rules that would convert the latest code from Delphi to C#, these rules were run quite frequently. During the project, we would get daily updates to the Delphi source. While each daily update was typically fairly small, its effects in C# could be much larger (e.g., adding a default argument to a method in Delphi only changes a couple lines of code, but may affect hundreds of call sites in C#). Therefore, it was imperative that the SmaCC Transformation Toolkit be able to retranslate a large software project quickly.

IV. Partitioning the Work

The migration project used four full-time developers: two consultants that had experience with previous non-Delphi migrations and two ProfitStars application developers. One of the application developers was on permanent assignment throughout the project, but the other position rotated among the rest of the development team as their expertise was needed.



Input:

```
  { Reset the variables and pause }
  Ball.Left := 388;
  Ball.Top := 172;
 end
 else
 begin
  { Bounce off of walls }
  if (Ball.Top <= 0) or (Ball.Top + Ball.Height >= pnlTable.Height) then
   Vertical := Vertical * -1;

  { Check for collision with paddles and reset tradjectory in opposite direction}
  If (Ball.Left <= 26) and (Ball.Top >= Paddle1.Top) and (Ball.Top <= Paddle1.
   Horizontal := Horizontal * -1;

  If (Ball.Left >= 662) and (Ball.Top >= Paddle2.Top) and (Ball.Top <= Paddle2
   Horizontal := Horizontal * -1;

  Ball.Top := Ball.Top + Vertical;      //Move the ball
  Ball.Left := Ball.Left + Horizontal;  //
 end;
end;
```

Output:

```
   /* Reset the variables and pause */
   Ball.Left = 388;
   Ball.Top = 172;
  }
  else
  {
   /* Bounce off of walls */
   if ((Ball.Top <= 0) || (Ball.Top + Ball.Height >= pnlTable.Height))
    UnitUntPong.Vertical = (sbyte)(UnitUntPong.Vertical * -1);

   /* Check for collision with paddles and reset tradjectory in opposite directio
   if ((Ball.Left <= 26) && (Ball.Top >= Paddle1.Top) && (Ball.Top <= Paddl
    UnitUntPong.Horizontal = (sbyte)(UnitUntPong.Horizontal * -1);

   if ((Ball.Left >= 662) && (Ball.Top >= Paddle2.Top) && (Ball.Top <= Padc
    UnitUntPong.Horizontal = (sbyte)(UnitUntPong.Horizontal * -1);

   Ball.Top = Ball.Top + UnitUntPong.Vertical;      //Move the ball
   Ball.Left = Ball.Left + UnitUntPong.Horizontal;  //
  };
 }
```

Rewrite	Node	Location
if	DelphiIfStatementNode	3024 .. 3126
binary expression	DelphiBinaryExpressionNode	3027 .. 3088

Rewrites

Files: 3

Fig. 3. SmaCC Transformation Toolkit

This also had the effect of exposing the entire development team to the translated code so the final cut over would go more smoothly.

The project was broken into 11 milestones that could be independently tested and monitored for their progress. They had a logical progression from simple external utility programs to the main software assets. The first eight milestones were either console-based utilities or simple GUIs. This allowed us to create the core rules that dealt with the language itself and some of the simpler libraries and GUIs without having to deal with complex interfaces.

To keep some of the initial milestones from including too much source, we had to break some dependencies. We may have needed a particular method or class in some unit (file) for the milestone, but we did not need all classes and methods in the unit for the milestone. If we included all classes and methods, then we would have needed to include several other units in the milestone even though they were never used by the actual code executed by the milestone. To fix these issues we either refactored the Delphi to break the dependencies or replaced some methods in translation. For example, if a method wasn't used for a milestone we could replace it in translation with a C# method that threw an exception. Replacing the method in C# would break the dependency on code that had not been translated. As the project progressed, these replacement methods were removed so that their Delphi source could be translated to C#.

One of the biggest, project-specific, jobs that needs to be performed is determining the mapping between library classes and methods from the old system to the new system. For example, a Delphi `TToolBar` from the `ComCtrls` unit maps to a C# `System.Windows.Forms.ToolStrip`. At the start of the project we generated a spreadsheet that contained all of the external types and methods used by the program. This spreadsheet was used throughout the project to map these types and methods to their equivalent C# types and methods. Since the data was stored in a spreadsheet, everyone in the project could use it to enter mappings as it required almost no training. At the beginning of the project, the ProfitStars application developers were more qualified to enter these mappings since they were experts in Delphi and had some experience in C#, whereas the migration consultants had very little Delphi experience.

One common problem that arises in migrations is that there is not a direct one-to-one mapping between a library in the old language and a new library. These cases are typically handled either by extension/helper methods or custom components. Extension methods are generally used when a single method call in the old system maps to multiple calls in the new system. While we could use transformation rules to map the one method call into multiple method calls, it is generally better to not duplicate these calls throughout the code base. When the differences between the old and new libraries are more complex, we typically write a custom component to behave as the classes did in the old system. This is commonly done with GUI components. Even though there are often what appear to be one-to-one replacements in the new GUI framework, each GUI framework has its own idiosyncrasies and the correct behavior of the application often relies on them. The replacement components range from simple subclasses that override a single method, to complete replacements where the entire functionality is custom-coded. For the PROFITstar ALM project, the extension methods and custom components added around 2,400 lines of code.

Sometimes creating special transformation rules for a particular method required more work than the benefit that you would get from the rules. If the C# method differed significantly from the Delphi, it was easier to specify a drop-in replacement C# method instead of the transformation rules. For example, transforming a Delphi method with inline assembler to C# would likely need several transformation rules, and those rules would not be very reusable. In these cases we used *method replacements*. Method replacements insert the supplied C# method in place of the original Delphi method. When the replacement is made, the Delphi code is checked for changes that have occurred since the C# replacement method was created. If any Delphi changes were made, a comment is added to the C# method so that we can review the changes made to the Delphi code and make corresponding changes to the C# code. This protected us from cases where the replaced method had been changed in the original code. Since the replacement was not pattern-based, without this notification, the replacement method would simply ignore any changes that had been made to the original code base since the replacement was written. Obviously, we strove to minimize the number of replacement methods that we used.

While most of the conversion simply used transformation rules on the existing code, sometimes the Delphi code was refactored beforehand to make the transformation easier. While we could have performed many of these refactorings using transformation rules, performing them in Delphi allowed us to verify the Delphi code before the C# code was runnable. The most common Delphi refactoring was changing constructors. In C# constructors no code can occur before the super constructor call. However, in Delphi, the super constructor call can occur at any time in the constructor method. Code that occurred before the super constructor needed to be moved so that it occurred as an argument of the super constructor call or after the super constructor call. Sometimes this required creating new constructors or methods and nearly always required assistance from the Delphi development team.

Another common Delphi refactoring was converting records into classes. While C# has support for records, pointers to records are only supported in unsafe code. Sometimes the records pointers were only used for optimization, passing a pointer on the stack instead of the whole record. However, many times they were used to pass a mutable reference to the record. In these cases, we converted the records into classes. Whenever a record pointer was assigned or passed on the stack, an object reference could be used instead. If the record was assigned or passed without using the pointer, then we needed to clone the object to keep the original record semantics. To assist with this refactoring, we wrote some special Delphi-to-Delphi transformation rules.

V. Experience

In this section, we point out the features of this project which contributed to its success and pitfalls that made the project more difficult than initially anticipated.

A. The Good Parts

1) Original Developers Available: Despite portions of the code base being nearly 20 years old, many of the original developers of the software were available to us. This was critical in many cases where we were translating a particularly convoluted piece of code. Often the developers knew the reasons for the complex code. Sometimes it was simply working around a bug in a library and as such could be eliminated in C#. However, other times it was handling a case that we had not considered. Having such resources available eliminated considerable time that would have been required to research the code.

2) Developers were familiar with C#: For many migration projects the developers must be trained in the target language. However, for this project the developers had already implemented some smaller projects in C#. This minimized the amount of retraining that the development group had to go through before they switched to the translated project.

3) Able to Change Original System: The team wasn't afraid of making Delphi changes to make the translation process easier. For example, Delphi has two ways of dealing with objects; pointers and interfaces. Most of the project's code had been using the more modern, interface, approach. However, there were still several of the old objects lingering around. The development team refactored those occurrences into the new form so the translation would be simpler.

However, this worked against us on occasion. Sometimes when a problem was identified, we handled it during the translation process. Communications got crossed, and the team made changes to fix the original code. This duplication of effort wasted time.

4) On-Site Customer: We had an on-site customer to give the final word on which features were necessary. Some features, mostly in the UI, were difficult to duplicate in C#. The on-site customer could tell us what aspects were required and what ones could be modified to better fit the libraries that we were using.

5) Similar Language Features: Both Delphi and C# are statically-typed, object-oriented languages. This eased the translation between the two because most of the language features from Delphi were present in C#. Where language features were not present in C#, (e.g. arrays with non-zero base indexes), we could create library classes to mimic their behavior.

6) Frequent Regeneration: The SmaCC transformation engine is fairly efficient and can be easily parallelized. Since we were creating transformation rules and not directly editing the generated code, we would make several rule changes each day. After each change, we would regenerate the problem file. Normally, it was only a matter of a few seconds to regenerate a file.

We also regenerated the complete code base on a daily basis. This allowed us to regression test the day's rule changes. We would verify the C# code changes to ensure that the rule changes were valid. Daily regeneration also allowed QA to test features in C# that were added to the original Delphi on the previous day. In fact, we occasionally had bug reports opened up when we failed to regenerate the source overnight and did not pick up a particular enhancement that had just been added to the original code. At the point of cut over, the two code bases had been in sync for months.

7) Global Changes: The rules allowed us to make global changes up to the end of project. In one example, we were using a GUI control that did not provide all of the necessary features. We did not find out about the required features until late in the project. Having rules allowed us to adjust the mappings to another control and regenerate the code. If we weren't using rules, it would have taken much longer to manually edit the code.

8) Original Language Expert: Every language has its own idiosyncrasies and corner cases that only the gurus know. In a code base as large at the one for this project, it is inevitable that some of these will crop up. For example, Delphi could pass a single integer as an argument to a function that expected an array of integers, and it would work as if we had passed a one element array. At least one of ProfitStars' developers was a Delphi guru and could explain how these things actually worked under Delphi.

9) Extensive Automated Tests: Anytime a change of this magnitude occurs, it is common for some new bugs to be introduced. Tests are useful for limiting the quantity of new bugs. Even though applying the transformation rules is an automated process, the fact is, humans are writing the rules and errors do occur. By having automated tests, we could validate some of the translation without having to involve the QA department or domain experts. The Profitstars code had over 13,000 automated DUnit[2] tests. Each project milestone had a collection of tests that were also migrated to use the NUnit[3] framework and had to pass for the milestone to be completed. Upon completion of the project all tests had to pass under C#.

[2]http://dunit.sourceforge.net/

[3]http://www.nunit.com/

10) Milestones: Breaking the project into smaller milestones had some advantages. First, having milestones helped track the overall progress of the project. While we could have used some other metric for measuring the overall progress such as "percent of code migrated", many times these measurements aren't that reliable. For example, a developer may consider the code migrated when it compiles or when it passes the unit tests, but QA may only consider it migrated when it passes all of their tests in addition to the unit tests. The explicit milestones let everyone from the developers to management know what the current status was.

The milestones also helped the developers remain focused on immediate problems for the current milestone instead of other less pressing issues. For example, part of the final milestone was C# code cleanup to eliminate some compiler warnings. Some of these warnings were present in the original Delphi code and others were created from the migration process. Since they did not affect the running of the code, they did not need to be removed until the end of the project. By having the C# code cleanup as part of the final milestone, we were able to focus solely on making the code work in the earlier milestones.

B. The Hard Parts

1) GUIs: User interfaces are generally the hardest part of any migration project. Often it is quite difficult to replicate the same look and feel between the two systems. Sometimes these differences are acceptable, but other times the original behavior must be duplicated even if it isn't consistent with other uses in the application or consistent with the operating system. Furthermore, most projects have few, if any, GUI tests that are automated or even written down.

GUI events are another big issue with migration projects. There are differences between the types of events and when these events are triggered. One of the most common differences is whether an event can be signaled when a property is set programmatically or only when the end user interacts with the control. For example, the Delphi list box signals its selection changed event only when the end user selects something using the mouse or keyboard. However, in C# the selection changed event is also signaled when the code explicitly sets the `SelectedItem` property. In some cases these extra events don't cause any problems. However, in others they can cause infinite recursion or more subtle differences.

2) External Library Design Differences: Large projects depend on several libraries. These libraries may be provided by the language itself or by third-party vendors. In either case, a replacement must be found in the target language. In most cases once the target is found, it is a simple matter of mapping the original library to the target library. However, in some cases it is much more complex. If the original and the target have different architectural styles, it may require making several changes to the migrated code. One of the larger differences for this project was the reporting library. In Delphi, the reporting library was event-driven, that is, each band in the report generated an event when it was about to be rendered on the page. The original program used these events to fill in the report controls with appropriate data for that band in addition to manipulating the control geometries and visibilities. The library we used in C# took a standard dataset and used that to generate the entire report. In Delphi, the program code was in complete control. It decided what to print and where to print it. However, in C#, the library was in control of these things. The code simply provided the report format and the report dataset. This difference required extensive scaffolding to support the old model on top of the new model and was a source of many of the last defects in the project.

3) External Library Bugs: Both the platform you are migrating from and the one that you are migrating to have bugs. The difference is that the developers know the bugs that are in the original platform and have worked around them. They don't know the ones in the target platform, yet. Working around these new bugs is often a time-consuming process. Furthermore, some of the work arounds for the original platform may not work for the target platform or may no longer be necessary and may need to be removed.

4) "Accidentally" working code: Several times on this project, we ran across code that worked under the old system, but broke under the new. Quite often, the Delphi code was arguably incorrect, it just happened to work due to some fluke in the Delphi runtime. When this was translated into C#, the code no longer worked because of the different runtime. These cases were often fixed by simply having the developers correct the original code. In an environment where developers resist this type of change, the translation would be more difficult.

5) QA Testing: Even though the project had several thousand automated tests, it still needed integration and system testing. The QA department provided this support. They were required to sign off on each milestone for it to be considered complete. In addition to obvious bugs, they were responsible for reporting any differences between the Delphi and C# programs. These differences would then be looked at by the on-site customer to determine if the difference was acceptable. Given the size of the project, this was a very time consuming process. It takes considerable time to test 1.5 MLOC. Furthermore, it takes even longer when you are looking for any differences between the two programs.

6) Refactoring During Migration: Just as it is generally a good idea to separate refactoring from adding features, it is also a good idea to separate refactoring from migration. While both refactoring and migration try to keep the behavior the same, they are different actions. For this project, the original Delphi code used a few different grid controls. We wanted to use a single grid control in C#. Instead of refactoring the Delphi code to use a single control before migrating, we decided to refactor while migrating. As a result, we didn't know what the required behavior of the grids was. For example, is the grid line color a required part of the behavior, or can it be ignored for the purposes of the refactoring? Instead of addressing these issues in a smaller refactoring step, we addressed them during the migration. Not only did this make the transformation rules more complex, it also made it more difficult to test since we had not yet determined what the required behaviors were.

7) Memory Management: Memory management in Delphi is a mix of explicit management and reference counting. Whenever the reference count reaches zero, the object's destructor is called. In C#, everything is garbage collected. When the object is garbage collected, the object's finalizer is run if it exists. There is no order guarantee on these finalizers. For example, a finalizer may need to write some data to a file and then close the file. However, since the file may have been garbage collected too, its finalizer may have already been run and closed the file. Therefore, when the finalizer tries to write data to the file, it will fail. Furthermore, there is no timeliness guarantee for finalization so a resource may not be released in a timely manner. For most objects, these issues were not important. However, for a critical set of objects, these behaviors mattered and introduced a nondeterministic set of errors into the translated code.

To deal with this, we added logging code to the finalizers. If an object finalizer was called without first calling the `Dispose` method, it would be logged. This logging was integrated into the unit tests. If any finalizers were logged during the test, the test failed.

Using this log information, we were able to isolate possible memory release issues and examine them on a case-by-case basis. In most cases, there would never be an error. But in others, we needed to refactor the original code to avoid sequencing issues during finalization.

8) Floating Point Numbers: Delphi has single, double, and extended precision (80-bit) floating point numbers. C# only has single and double precision floats. Code that used extended precision numbers was converted to use doubles. This caused some differences in the calculations. This was especially troublesome in cases where no delta was taken into account when comparing two floating point numbers for equality. Where it caused the most difficulty for us was in getting reports to match exactly. For every difference, we had to bring in the domain expert from the development team to determine if the difference was insignificant, or was the result of an actual error. This was exacerbated by the fact that some of the regression test data contained unrealistically large numbers.

9) Managing expectations: When we were in the initial phase of this project, we had told management that the translated application would work "just like the original." After some of the early milestones, the client remarked that they were impressed with how closely the translated version resembled the original. However, we became victims of our early success. Towards the end, we were getting defects opened up against the translation because certain screens were redrawing more visibly than they were before the translation. The screens were virtually identical to the originals, they just painted more slowly, in some cases, than they had in the original system. During the process, expectations had risen to a level that was difficult to achieve. Fixing many of these types of things lie outside the scope of direct translations. Different systems have different performance characteristics. Some operations will be faster. Others will be slower. The appropriate time for dealing with many of these issues is after the completed translation by working directly on the translated code.

10) Visual Studio: Since this was a C# project, we used Microsoft's Visual Studio development environment. However, this presented some problems. First, unlike the Eclipse Java IDE, Visual Studio does not allow running a project that has compile errors. In Eclipse, methods that have compile errors simply throw exceptions if they are executed. This behavior makes migration projects much nicer as you can start testing the migration earlier than we could using Visual Studio. With Visual Studio, most of the transformation rules had to be written before we could test any of them.

Another issue with Visual Studio was compile times. Compiling the entire project would take about 3 minutes. This added considerable time to debugging as making a small change could require another 3 minute compile. While edit-and-continue helped eliminate some of these compiles, it has several restrictions on when it works. For example, you cannot use edit-and-continue if the method contains an anonymous method.

One last problem was getting the .cs designer files in a format that Visual Studio's GUI designer could edit without introducing new bugs. While the designer files are simply C# code, they have some restrictions on some of the code constructs and methods that can be used. If you used one of these items, the designer might open without any problems, but upon saving the interface, certain properties would be quietly lost.

VI. Summary

Switching implementation languages for a legacy project is a difficult, time consuming, and costly task. By performing the translation using a rule-based, transformational approach, the translation can be performed in parallel with actual development. This minimizes the impact on the development process and provides a viable path for a project to modernize their code base.

References

[1] J. Spolsky, *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress, 2004.

[2] D. Roberts and J. Brant, "Tools for making impossible changes," *IEE Proceedings – Software*, vol. 151, no. 2, April 2004.

[3] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon, "TMM: Software maintenance by transformation," *IEEE Software*, vol. 3, no. 3, pp. 27–39, 1986.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS®: Program transformations for practical scalable software evolution," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 625–634.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.17. A language and toolset for program transformation," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 52–70, 2008.

[6] J. R. Cordy, "The TXL source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, 2006.

[7] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, pp. 253–263, 1997.

[8] W. Loew-Blosser, "Transformation of an application data layer," in *OOPSLA '02: OOPSLA 2002 Practitioners Reports*. New York, NY, USA: ACM, 2002.