AUTOMATED TRANSLATION OF LEGACY CODE FOR ATE

Andrew Moran Jim Teisher Andrew Gill
Galois Connections Inc.

Emir Pasalic
Oregon Graduate Institute

John Veneruso
Credence Systems Corporation

Abstract

When an Automated Testing Equipment (ATE) company designs a new system, the issue of backward compatibility is always a major concern, both for the company and its customers. If backward compatibility is maintained, the ATE application engineers face the difficult task of trying to support new features on an aging system. The alternative is to face the problem of converting old test programs to the new environment. Translation of legacy code involves an automatic translation tool, and some application effort applied to those problems the translator couldn't resolve. To minimize the amount of work required from the application engineers, the tool needs to be semantically-aware; that is, the tool must contain domain-specific knowledge and use that knowledge when translating. The more knowledge a tool has at its disposal, the less code an application engineer is forced to translate by hand.

Until recently, it has been difficult to perform automatic translation satisfactorily because it was not cost effective to write a translator that possessed such semantic understanding of the test programs. By making good use of Functional Programming techniques and tools, we were able to construct a cost-effective, semantically-aware translation tool in a fraction of the time needed by traditional methods. Based upon its performance during testing, we believe the tool to correctly translate the majority of test programs, thereby greatly easing the applications engineers' burden.

1 Introduction

Credence Systems Corporation (Credence) has a long tradition of maintaining 100% backward compatibility when releasing new software. Test programs written 10 years ago will execute on the latest version of the Quartet test system. Customers have been saved from needing to port test programs when a new system or release came out, but there has been a cost. The customers are forced to use an aging Application Program Interface (API), and the kinds of optimizations available to Credence software developers are greatly restricted.

Credence recently made the decision to migrate from SunOS 4.3 to the Solaris. This in turn required a change from Kernighan & Ritchie (K&R) style C [KR78] to ANSI C [KR88, ISO90]. This was forced upon Credence because the C compiler used on the original configuration is not available under Solaris. The C compiler chosen for the new configuration is the GNU C Compiler (`gcc`), which compiles ANSI C. While `gcc` can be configured to be very forgiving of non-ANSI code, it is nowhere near as lax as the original C compiler. In addition, the benefits of requiring ANSI-compliance are great. Credence also took this opportunity to make significant changes to the Automated Testing Equipment (ATE) API; in particular, the central data structure was made into an abstract datatype, in order to give Credence software developers more freedom in optimizing the internals.

Committing to these changes presented Credence with an age-old problem: how to cope with legacy code? Customers' test programs now need to be updated:

- They need to be Solaris-compliant;

- They need to be ANSI-compliant, and
- They need to use the new API.

Care must be taken to minimize the impact to customers, while preserving the readability and correctness of their code.

This paper describes the automated translation tool that Galois Connections Inc. (Galois) developed to solve Credence's legacy code problem. By building upon an existing C parser and analysis toolkit written in the functional programming language ML [MTHM97], Galois was quickly able to construct a translator that could cope with all three of the points above. The output of the Translator is a list of necessary changes in `diff` [MANa] format. This `diff` file is passed to the UNIX `patch` utility [MANb], producing the translated program. Using the `diff` format for changes has many advantages:

- Only those parts of the original program that *need* to be changed get changed, so impact on customer code is minimized.
- The `diff` file provides an explicit log of the changes that are proposed.
- `patch` can also use the Translator's `diff` output to reverse the translation.

The Translator does not constitute a 100% solution; there are bound to be cases that the Translator cannot cope with. In such cases, a Credence Applications Engineer will be needed to do the translation by hand. In practice, we expect the tool to correctly translate more than 95% of test code.

The rest of the paper is organized as follows. Section 2 describes in more detail the requirements of the translator, gives examples of the kinds of translations required, and discusses the issues that arose as a result of those requirements. Section 3 presents the basic idea of Galois' solution, and describes in some detail the translation process. In Section 4, we discuss the Functional Programming technology that underlies the Translator.

As the tool is still being readied for deployment at the time of writing, accurate figures for performance are not available. Section 5 presents some figures based on the Translator's performance during testing so far, and Section 5 concludes.

2 The Problem

There are two fundamental requirements for the translation tool:

- Translated code must compile correctly on the new system.
- Code changes should be as minimal as practical. Translated code must preserve macros, comments, and existing code structure.

The first is a correctness criterion; the second ensures that the upgrade has minimal impact on Credence's customers. All of the challenges arising from this project stem from the fact that there is significant tension between the two requirements.

The first requirement could most easily be satisfied if the translator code acted upon *post-processed code*; that is, code that has been run through the C preprocessor `cpp`. This is the code that C compilers actually work on: all comments have been removed and macros expanded. But doing so would violate the second requirement.

In order to satisfy both requirements, the translator must produce pre-`cpp` code. This means that the translator must be aware of comments, `#define`'d macros, `#ifdef` structure, and `#include` files, preserve them in the output, *and* ensure that the result compiles on the new system.

2.1 Compilation Requirements

Broadly, the requirements stemming from the need to have code compile with `gcc` on the new system may be divided into three categories:

- Translated code must compile under Solaris;
- Translated code must be ANSI-compliant (or close to), and
- Translated code must use the new API.

We will look at each of these in turn.

Solaris Most of the differences between SunOS 4 and Solaris lie in aspects of the operating system itself (such as signal mechanisms). The test programs tend not to be heavy in their use of OS primitives, so this

aspect of the translation is simpler than it first appeared. However, the change to Solaris has introduced many new system function names, such as `signal` and `clock`, which may collide with names in test programs. These names are common in test programs, and they must be renamed under Solaris to avoid conflict.

ANSI There are major differences between K&R C and ANSI C, too many to detail here. Matters are further complicated in this case because the original C compiler was a very lax implementation of K&R C. Programs that would be in error with other K&R C compilers would pass unscathed through the original C compiler. Among the most common problems in this category are variables and functions being used without the relevant `.h` file being included. ANSI C also introduced the concept of function prototypes, which are a more detailed kind of function type definition not available in K&R C. ANSI C compilers use function prototypes do perform more stringent type-checking. The fact that the original C compiler was so lax in its type-checking only exacerbates the situation. The difference between the way K&R and ANSI deal with `#define` macros, coupled with some very imaginative macro usages, presents a very interesting challenge (see Section 2.2 below).

New API Credence's API is based around a data structure called a `PINLIST`. In the original API, it was defined as a typedef to an unsigned short, and this information was visible to test programmers. The new API makes that structure opaque. Thus, any test program that used pointers to unsigned shorts instead of explicitly defining `PINLIST`s, pointer arithmetic over `PINLIST`s, or other operations that are possible because the inner structure of `PINLIST` is known, will break under the new API unless changed.

2.2 Example Translation Issues

In this section we give some examples of the kinds of problems the Translator has to deal with.

Unit Macros. A common use of macros in test programs is to have macros denoting units, as a form of documentation:

```
#define MS      *0.001
#define MA      *0.001

...

... func34 (20MA, 300MS);
```

Here, `20MA` is intended to mean "20 milliamps", and `300MS` is intended to mean "300 milliseconds". In K&R C, these work fine, since macro expansion acts at the string level. In ANSI C, macro expansion acts at the lexeme level. Consequently, since `20MA` and `300MS` constitute single lexemes. The problem is that while `20*0.001` is legal ANSI C syntax, the single lexeme `20MA` is not, and is reported as an error by `gcc`.

Functions whose return type has changed. There are a number of functions defined by the Credence API, but not implemented by it; they are intended to be implemented in the test program. Credence made a few small changes to these functions a few years ago. These changes had no impact to customers while they were using a K&R C compiler.

Since the original C compiler didn't require a function's definition to be in scope before that function was used, many customers didn't bother including the headers that defined these functions, and as a result the return type of these functions defaulted to `int`. This allowed test programmers to return values from these functions, even though those values would always be ignored:

```
test1 () {
    ...

    return SUCCESS;
}
```

These functions all return `void`. The return statement above would yield an error under `gcc`, so the Translator must replace the statement with a simple return.

Opaque PINLIST. The original definition for `PINLIST` was a typedef to unsigned short. Occasionally, for a variety of reasons, customers decided to use unsigned short instead of `PINLIST` when defining variables. This would lead to situations such as:

```
unsigned short pl[] =
    {1, 2, 3 | LAST_PIN};
int someUnrelatedVariable;
 ...
func_test(pl, 0, 10);
 ...
```

Here a function in the Credence API, `func_test`, is being called. The function expects a `PINLIST` as its first parameter. This works fine under the old API, but the new API defines `PINLIST` as a structure. In order for this code to function as intended under the new API, a number of changes must be made. Since `pl` is being passed into a function expecting a `PINLIST`, its definition must be changed. This is more than just a simple textual substitution, however. Since `pl`'s original static definition is incompatible with `pl`'s new type, it must be explicitly initialized. The resulting translation is:

```
PINLIST *pl = pinlist_init();
int someUnrelatedVariable;

pinlist_addPin(pl, 0, 1);
pinlist_addPin(pl, 1, 2);
pinlist_addPin(pl, 2, 3);
 ...
func_test(pl, 0, 10);
 ...
pinlist_destroy(pl);
```

The variable `pl` has been changed to a `PINLIST`, and the creation function has been called. The static values are also assigned to the structure. Function `func_test` is called as before, but now its parameters are type correct. Finally, the `PINLIST` is destroyed before the function exits.

3 The Solution

The Translator works by parsing C source files one at a time, analyzing them to discover what changes are needed, if any, and then outputting a list of changes required. This section explores that process in more detail.

The translation process has two phases. The first builds a database of known entities, and the second analyzes a source file, using that database to build a list of required changes.

The initial phase involves reading in all relevant header files. These include all Solaris system files, all external Credence API files, and all internal Credence API files. Every variable, function, type, and macro that is defined in these files is entered into a database. For each entity, we record its name, type, header file of origin, and which of the three categories above it belongs to: Solaris, External, or Internal. The distinction between external and internal Credence header files is important, because internal functions are not supported, and are not intended to be used by the test programmer. The Translator warns of such usages, but does not remove them.

Other sets of header files may be added to the database, as needed. For example, a customer site may have a common testing library of their own that their test programs use. The Translator will be deployed with a database already populated with standard Solaris and Credence header files, so that users need only add those headers that are specific to their site or project.

The second phase of the translation proceeds by reading in a single source file. The file is parsed: translated from text into an abstract syntax tree, a data-structure enabling analysis of the source file. This is similar to the second phase of a C compiler, which follows application of the preprocessor. In fact, the Translator's parser is a little more sophisticated than that of most C compilers, because of our special needs and the fact that we need to parse `#`-directives as well.

The heart of the Translator is a suite of functions that walk over the abstract syntax tree and generate changes that must be made. Each of the tasks mentioned above has a corresponding function. Some of the functions are simple, like the one that discovers when a return value appears in a function that returns `void`, and changes the return statement into a simple return. Others, like discovering when a header file needs to be included, require more analysis. The most complex of all of the functions is the one responsible for the `PINLIST` translation. It incorporates a clever type inference algorithm, which is capable of discovering when variables not declared as `PINLIST`s are in fact used as such.

The changes generated by all of these functions are fed into a difference engine, which composes all of the changes and outputs them in `diff` format. Typical output looks this:

```
2a3,9
>
>
> #include "box.h"
> #include "summary_protos.h"
> #include "shutdown_device.h"
>
>
3c10
< int initialize_tester(){
---
> void initialize_tester(){
4c11
<   return 99;
---
>   return;
```

The UNIX `patch` utility understands this output and uses it to transform the original test program into the a program that uses the new API, and compiles under Solaris with `gcc`. The notation `2,3a9` means that the following 8 lines will be added to the original program. In this example, three `#include`s were needed. The next two patch directives result in changes reflecting the fact that `initialize_tester` has been changed to return `void`. Lines prefixed by "<" are the lines in the original that are to be replaced; lines prefixed by ">" are the lines intended to appear in their place.

All of the changes derived by the Translator are presented in this easy to understand format, enabling applications engineers and customers to readily see what changes are proposed (or have been enacted). The same file can also be used to undo any changes, should they be erroneous.

4 The Underlying Technology

In this section we discuss the possible approaches to the problem, and explain some aspects of Functional Programming, the powerful technology behind the Translator.

4.1 Possible Approaches

There are at least three possible approaches to generating the code changes required to allow the migration from SunOS 4.3 to Solaris and enforce the use of the new opaque `PINLIST` API.

Firstly, we could provide detailed guidelines to allow clients to translate their programs by hand. This may be acceptable if the number of programs were small, but the installed base of test programs is in the order of tens of thousands. It was simply not feasible to expect clients to change so many lines of their legacy tester code.

Secondly, we could provide scripts (written, for example, in `perl` [Vro96]) that make syntactic changes to client programs. In the past, Credence has used such scripts to translate programs from one version to the next, but the current translation task is much more complex than any attempted previously. The difference between K&R C and ANSI C is profound, and the task of converting programs that had used an open datatype to the use of an abstract one is daunting. This is beyond what can reasonably be done with a scripting language.

Thirdly, we could provide a semantically-aware translator. It became obvious very quickly that such a translator was needed; the problem quickly became one of where to find such a beast. The first attempt looked at modifying the open source ANSI C compiler `gcc`. There are some fundamental problems with using such a compiler to provide a translator.

- Compilers have some of the required level of semantic understanding, but compilers are complex to modify, requiring specialized knowledge of their internal algorithms. A senior engineer with some experience with compilers spent a month working on modifying `gcc` with very little success.

- Compilers are written to understand program with the aim of producing an executable image, not outputting the original programs with modifications. Compiles pay no attention to things that matter to human users, like comments and layout of code. If no one was ever going to need to read or maintain the modified program, a compiler would make a solid base for a translator. However, making the result readable interferes severely with the functional changes required.

- Even having overcome these difficulties, the major change of making `PINLIST`'s structure opaque

requires knowledge of analysis systems powerful enough to recover the underlying abstraction.

Through some research Credence discovered that there was a technology that was capable of providing a semantically-aware translation: Functional Programming.

4.2 Functional Programming

Functional programming languages have been under development in universities for the last twenty years, and are now ready for prime-time. They provide many features that are directly relevant to the legacy code problem:

- Tree-manipulation is the bread and butter of functional languages. They provide very concise methods for specifying the structure of recursive trees, and for describing passes over trees which collect information, and which can generate new versions of the trees. This is directly relevant to code transformers because, after parsing, code is represented as an abstract syntax tree. Writing analysis passes is straightforward, even when very complex or subtle information is being gathered. Functional languages are not appropriate for all tasks, but of all the application for which functional languages have been demonstrated, handling and manipulating computer programs has consistently been the most popular.

- Functional language programs are typically 1/10 to 1/5 the size of corresponding programs written in languages like C++ or Java, and this translates directly to a productivity increase. It has often been said that a well-written function program reads like a design-document, and that programming in the functional language is design-level programming. This raises the question as to how all the implementations details get added. The answer is that the functional language compiler provides them. Often it can do a great job, and produce code comparable in efficiency to C++ code that might take days to produce by hand. In other cases, the code may be significantly less efficient. However, when the code is not speed-critical, the trade-off between code-efficiency and programmer productivity should be weighted towards productivity. The ensuing reduction both in software cost and in time to market are compelling.

  Building upon the SML/NJ C-Kit [HOM], a C front-end for the functional language Standard ML [MTHM97], Galois was able to produce a translator prototype[1] in just four weeks. The prototype's core modules were of sufficient quality that they comprise the core of the finished tool. We estimate that the same task, using traditional methods and building upon a similar toolkit for manipulating C code, would have taken at least 3 months.

- Functional programs make strong use of powerful type systems, and Galois' engineers are experienced in both the design and implementation of type systems in functional languages. The techniques and experience of implementing type systems was directly applied to the analysis required to recover the abstraction behind the `PINLIST` structure.

Galois chose the functional language Standard ML primarily because of the existence of the C-Kit and the fact that Pasalic had done significant work in Standard ML using the C-Kit to implement a prototype `PINLIST` analysis. Other functional languages, such as Haskell [PJHA$^+$] and OCaml [Ler], would also have been suitable.

4.3 Other translators

AnnoDomini [EHM$^+$99], produced by Danish software company Hafnium ApS, used a complex type system to find Y2K related shortcomings in software. The tool was implemented using the ML Kit [BRTT93], a Standard ML compiler with some special memory management features. The authors claim that the project would have been impossible without the use of functional programming technology.

The translator FermaT [War99] is able to capture the entire functionality of an IBM mainframe assembler program, and produce an equivalent, maintainable C or COBAL program.

The Design Maintenance System[MB97] is a software engineering toolkit produced by Texas software company Semantic Designs. It is intended to support the incremental engineering and maintenance of large application sys-

[1] It was able to derive a list of header files needed to be included in a source file, discover when Credence API entities were shadowed by local definitions, and output the appropriate `diff` file.

tems, driven by domain knowledge, semantics, captured designs and automation.

The principals behind the latter pair of products have a background in rewrite systems, which are very closely related to functional programming.

5 Performance

The tool is being readied for deployment at the time of writing, so accurate metrics of its performance and time saved due to its use are obviously not available. However, we can give some figures based upon the Translator's behavior during the (ongoing) testing phase.

Recall that the Translator's operation may be divided into two phases: known entity database creation, and translation:

- Typically, when creating the database of known entities, the Translator processes between 1,000 and 2,500 header files. Our test suite contains around 2,500 header files, comprising around 100,000 lines of code. The Translator processes these files in less than 45 seconds on a Pentium III 900 MHz.

- We have tested the Translator on about 1000 files (some 100 created by Galois, for unit and regression testing; the rest being representative of real test programs, provided by Credence), comprising over a million lines of code. On that input, the Translator processes more than 300 lines per second on average[2].

The most important factor in judging the success of the Translator from Credence's point of view is time: how much time will be saved by using the Translator instead of some alternative? This is impossible to measure and even quite difficult to estimate. (It is worth noting that Credence's clients have many millions of lines of test code between them. Translation by hand is simply not an economically viable option.)

That said, we are sure that the Translator will be prove to invaluable. Credence had a team of engineers comb through the 900 files mentioned above, translating them to Solaris/ANSI by hand. This took approximately 6 engineer months[3]. The Translator can translate the same set of files in less than two hours. Of course, since the Translator is not a 100% solution, an applications engineer would need to work on some of those files. Just how much field intervention will be required remains to be seen, but we are confident that it will be within tolerable limits.

6 Concluding Remarks

The Translator is capable of successfully translating almost all test code, and the translated result is as easy to read as the original code. By making the `PINLIST` datatype opaque, Credence has given itself greater freedom to optimize the implementation of its system architecture without inconveniencing its customers. Their customers can now take advantage of state-of-the-art operating systems and faster testing environments with a minimum of fuss.

By using functional programming, Galois was able to produce a high-quality tool with a small team of engineers in a matter of months. Such techniques are not limited to the translation of legacy code. Galois uses functional programming as its core technology, designing executable models of complex systems, domain-specific special purpose languages, and powerful analyses.

Acknowledgments. Early work on the `PINLIST` component of the translator was done by Pasalic in cooperation with Professor Tim Sheard, of the Oregon Graduate Institute, and his help is gratefully acknowledged. Without the SML/NJ C-Kit, this work would have been more difficult, and would have taken much longer. We also indebted to Les Hemmingson and Randy Smith of Credence for their valuable input throughout this project.

[2] Processing time varies greatly between the Solaris/ANSI conversion pass and the `PINLIST` translation pass: the latter is an order of magnitude slower. This is mainly due to the fact that the Solaris/ANSI pass need not process header files if they exist in the known entity database, whereas the `PINLIST` pass must process them regardless, in order to derive accurate type information.

[3] The main task of the team was to discover *how* to do the translation, so this is an unfair comparison. But it's the only one we are able make.

References

[BRTT93] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML kit (version 1). Technical Report DIKU-report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993.

[EHM+99] P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sørensen, and M. Tofte. AnnoDomini: From Type Theory to Year 2000 Conversion Tool. In *ACM Symposium on Principles of Programming Languages POPL '99*. ACM, January 1999. Invited talk.

[HOM] N. Heintze, D. Oliva, and D. MacQueen. The **C-Kit**: an SML-NJ front-end for C. http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ckit/.

[ISO90] *Programming Languages – C*. International Standard. ISO/IEC, 1990. Ref. nr. ISO/IEC 9889: 1990(E).

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, first edition, 1978.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, second (ANSI C) edition, 1988.

[Ler] X. Leroy. Objective Caml. http://caml.inria.fr/ocaml/.

[MANa] *DIFF – find differences between two files*. UNIX man page.

[MANb] *PATCH – apply a diff file to an original*. UNIX man page.

[MB97] M. Merlich and I. D. Baxter. Mechanical tool support for high integrity software development. In *IEEE Conference on High Integrity Systems*, Albuquerque, New Mexico, October 1997.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. ISBN 0-262-63181-4.

[PJHA+] S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98 Report. http://www.haskell.org/onlinereport/.

[Vro96] J. Vromans. *Perl 5 Desktop Reference*. O'Reilly, 1996. ISBN: 1-56592-187-9.

[War99] M. Ward. Assembler to C Migration using the FermaT Transformation System. In *International Conference on Software Maintenance*, Oxford, England, August 1999. http://www.smltd.com/.