Journal of Software Engineering and Applications, 2013, 6, 30-40
http://dx.doi.org/10.4236/jsea.2013.64A005 Published Online April 2013 (http://www.scirp.org/journal/jsea)

Source-to-Source Translation and Software Engineering

David A. Plaisted

Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, USA.
Email: plaisted@cs.unc.edu

Received February 5th, 2013; revised March 7th, 2013; accepted March 15th, 2013

ABSTRACT

Source-to-source translation of programs from one high level language to another has been shown to be an effective aid to programming in many cases. By the use of this approach, it is sometimes possible to produce software more cheaply and reliably. However, the full potential of this technique has not yet been realized. It is proposed to make source-to-source translation more effective by the use of abstract languages, which are imperative languages with a simple syntax and semantics that facilitate their translation into many different languages. By the use of such abstract languages and by translating only often-used fragments of programs rather than whole programs, the need to avoid writing the same program or algorithm over and over again in different languages can be reduced. It is further proposed that programmers be encouraged to write often-used algorithms and program fragments in such abstract languages. Libraries of such abstract programs and program fragments can then be constructed, and programmers can be encouraged to make use of such libraries by translating their abstract programs into application languages and adding code to join things together when coding in various application languages. This approach can also improve program reliability, because it is only necessary to verify the abstract programs once instead of verifying them separately in each application language. Also, this approach makes it possible to generate code faster than programming from scratch each time. This approach is compared to the use of libraries and to other methods in current use for communication between programming languages and translation between languages.

Keywords: Source-to-Source Translation; Libraries; Legacy Code

1. Current Software Practice

Problems with producing and maintaining software are well known. For example, Hinchey *et al.* [1] write "While hardware dependability has increased continually over the years, and with mean time to failure (a measure of dependability) for the most reliable systems now exceeding 100 years, software has not kept up with this pattern and indeed has been exhibiting declining levels of dependability." Denning and Riehle [2] write "Approximately one-third of software projects fail to deliver anything, and another third deliver something workable but not satisfactory. Often, even successful projects took longer than expected and had significant cost overruns. Large systems, which rely on careful preplanning, are routinely obsolescent by the time of delivery years after the design started." The problems of buggy software in big data applications are also highlighted in [3].

The problem is exacerbated by the need to rewrite programs over and over again for different languages and machines. Such rewriting would not be necessary if it were possible to translate programs, or portions of programs, from one high-level language to another so that they would not have to be written from scratch in each language. This would be especially helpful for high-level imperative programming languages with side effects, arrays, and pointers or structures, because such languages tend to be efficient and are widely used. Programs that do such translation have been written, and are called source-to-source translators.

2. The Potential of Source-to-Source Translation

The widespread use of source-to-source translation for imperative languages would help software engineering if such translators could be written and if it were easier to translate an existing program in another language than to program from scratch. Under these assumptions, coding would be faster and the resulting programs would be more reliable because they would be more widely used. In addition, programs might last a lot longer than they currently do; if their original language became obsolete, they could be translated into other existing languages, and thus become effectively immortal, that is, their lifetime could be effectively as long as civilization continues in its current form. They could also be more widely used, because they could be translated into other languages. This would aid program reliability, because programs that last a long time and are heavily used tend to be more reliable. Furthermore, people would be more likely to write programs carefully if they knew that the programs would be preserved a long time and widely used.

Source-to-source translation has been studied by various researchers, and is often used together with program optimization. However, it has also been used even without program optimization, and in this way can still be of great benefit. Translators that do not optimize programs but simply preserve the same program structure from one language to another have significant potential for software engineering. Such translators are also easier to write and faster to execute than those that perform substantial program optimization. It is also important for the translation to preserve execution speed as far as possible, so that the resulting program is nearly as fast as the original. The translator also should translate correct programs into correct programs, and should be automatic, as far as possible. The translator may not work on all source programs, but it should work on as many as possible.

However, even if the translator is not fully automatic or fully verified, it can still be helpful. Requiring some human interaction in the translation process or requiring some coding can still be easier than coding everything from scratch. If the resulting program is not guaranteed to be correct, then it can be tested and debugged; if the program is close to correct, this may still be easier than writing it from scratch. Some language features may be difficult or impossible to translate automatically, but the translator can still be of use on programs or portions of programs not containing these features.

As an example of parts of languages that may not be translatable, Bates [4] writes "But in the following decade, the industry reversed course, choosing C and later C^{++}, which not only allow, but routinely require, highly unsafe methods scarcely above the assembly-language level, with huge regions of semantics that are explicitly disavowed as 'undefined.'" Such regions of language may not be possible to translate, unless subsets of them are better behaved.

Formally, one can define a program fragment to be a portion of a program that is a syntactic unit, such as a procedure, a statement, or a sequence of statements. Then associated with each source-to-source translation there would be a core, or set of fragments, on which the translation can be done. Of course, different translators may have different cores; the larger the core, the more complicated and time-consuming the translation process may be. It may also be convenient to define abstract languages, which are languages with a simple syntax and semantics, making it easier to translate the entire language into other high-level imperative languages. **Figure 1** illustrates translating an abstract language into *n* application languages. For the sake of efficiency and applicability, such abstract languages should be imperative languages with side effects, arrays, and pointers or structures.

Source-to-source translation could be of greatest benefit to programmers if program libraries were available having program fragments in various languages and translators into other languages. Associated with each program fragment would be a specification, or text description, and there would be a means of searching the library for fragments having a given specification or description. It might be helpful for the programmer to be able to specify the names of certain variables and procedures in the target program. Then programming would involve searching for program fragments in various languages, translating them to the same language, putting them together, possibly adding some code by hand, and testing portions of the target program at various stages in this process.

For this purpose, it is not necessary to have translators between every pair of programming languages. Instead, it may be possible to translate between two languages through a series of intermediates. For example, if there is an abstract language A with translations into and out of languages P_1, P_2, …, P_n, then it is possible to translate between any two languages P_i and P_j by going through A.

This only requires $2n$ translators to translate between n languages. Other arrangements are possible; for example, one can arrange languages in a tree structure, with translations in both directions along each edge of the tree. It is also possible to translate between n languages by putting them into a loop, but this may require many translation steps to get from one language to another.

3. Legacy Code and Algorithms

Source-to-source translation is often thought of in connection with legacy code, which is large application code written by industry or another organization that originates from an earlier software release. "These legacy

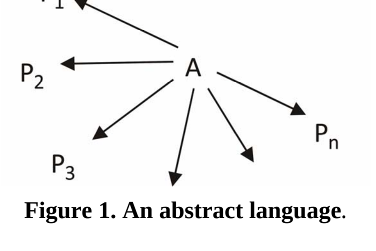

Figure 1. An abstract language.

systems often date from 1970's when the concepts of proper software engineering were relatively new and proper documentation techniques were not a concern" [5, p. 18]. Such code may represent decades of development effort [6, p. 343]. Frequently large portions of operating systems are legacy code from previous versions of the system. Much legacy code is still in COBOL or Fortran, languages that are not widely taught today. This code frequently contains errors and security vulnerabilities and is not well understood. Sometimes good documentation may be lacking, and sometimes the source code is not even used or has been lost and the code is run in binary. Legacy code may be expensive to maintain because it is poorly understood, runs on obsolete hardware, lacks a clean interface, or is difficult to modify. Legacy systems continue to be used by industry "because of the prohibitive cost of replacing or redesigning them, despite their poor competitiveness and compatibility with modern equivalents" [7, pp. 601-602]. Most of the time this code is run in its original language and interfaced to from more recent code, possibly by message passing. A system often used for this is the MPI message passing interface [8].

For large, poorly understood legacy code containing errors, source-to-source translation is of limited value. Translating such code into a more modern language does not eliminate the errors or make the code easier to understand; the translated code may even run slower and be harder to understand than before. However, sometimes it is necessary to modify legacy code due to changing requirements, and translation to another language may make the code easier to modify and maintain. Some legacy code is even in a current language but still may have errors and be poorly understood. Of course, for such code, source-to-source translation is most likely not helpful.

In addition to legacy code, which is used for application programs, there are many other programs that may be smaller, well-understood, well-structured, and well-commented, but for which it would help to translate the code into a currently maintained language. An example is a program written by students, or a general algorithm such as a maximum flow algorithm or programs for red-black trees. Such general programs or modules are not likely to have many operating system calls, goto statements, or unusual parameter passing methods, which can be difficult for source-to-source translation. General algorithms are likely to have a clean, simple specification and be used by many different persons and organizations, unlike legacy code, which is typically used by a single organization. General algorithms are also most likely written over and over again in many different languages or even in the same language by many different people. It is likely that a programmer, in the course of writing an application program, will use many such modules, and the resulting application program will call each module many times. Communicating with such modules or algorithms using messages, which is commonly used for legacy code, does not seem as desirable for such modules; it introduces inefficiencies and may be difficult to implement in some cases, such as recursion. Use of message passing may also make verification and compiler optimization more difficult.

These things make the cost tradeoffs for general algorithms different than for legacy code; because such general programs tend to be small and frequently used and called, it makes more sense to invest extra effort to make them translatable into many languages, as opposed to complex legacy code used by one organization. For widely used, general modules, source-to-source translation is a viable alternative to writing the program again from scratch in many different languages. Such widely used programs are the main concern in this discussion rather than legacy code. A better source-to-source translation methodology could even encourage the development and availability of many such widely used modules and thereby make programming easier. Still, if there are a large number of calls to legacy code, translation of this code into a currently used language would improve efficiency because the interface to the code would be faster.

It is possible that in the future people will decide to use message passing even for interfacing with general modules in other languages; it is hard to predict the course people will choose to follow. This approach would avoid the need for source-to-source translation for this application. In general, it is not reasonable to expect all programs to be written in the same language because different languages are suited to different applications. Therefore it is sometimes better to interface programs or algorithms in different languages instead of translating everything to the same language.

Another possibility would be to have a multi-language library with common algorithms written from scratch in many different languages. If one wanted an implementation of red-black trees in some language, for example, one could go to such a library, look up red-black trees, and find programs for them written in many languages. Of course, this approach requires more storage than having a single program and translators from it into many languages. Also, when a new algorithm is added to such a library, it requires considerable work to program it in many languages; if translators were available, the algorithm would only have to be coded once in a form suitable for translation into other languages. Furthermore, use of a verified translator would increase reliability because it would not be necessary to verify so many different programs that do the same thing in different languages.

4. Past Work

The source-to-source approach has been considered by a number of authors, and various systems are or have been in use.

Some systems that have been implemented or considered, use the source-to-source approach for translating between languages without attempting to optimize the code. For example, Wallis [9] considers automatic translation of other high-level languages to Ada. He mentions that various systems have been implemented or designed, but emphasizes their limitations. He considers that code may have to be redesigned to take advantage of the features of Ada, that translated code may be difficult to maintain, and that certain features of other languages may be difficult to translate into Ada. His emphasis is on totally automatic translation of an entire high-level language into Ada.

Albrecht *et al.* [10] discuss the translation between Ada and Pascal. They define subsets of each language and only translate between these subsets. These sublanguages were not easy to define. The translations preserve the semantics and structure of a program from one language to the other. The translations are mostly local in nature. They believe that this methodology can be applied to any two high-level languages.

Huijsman *et al.* [11] also consider translating Algol 60 into Ada. They note that there have been many attempts to translate programs between languages, even in 1986. They also note that it is difficult or even impossible to translate some Algol 60 constructs into Ada, particularly in the areas of goto statements, parameter passing, and interaction with the operating system. However, they note that "in a large number of cases large parts of the Algol 60 programs can be translated mechanically" (Huijsman *et al.* [11], p. 48). Overall, they found that 80 to 90 percent of Algol 60 code could be translated automatically. Sometimes a manual coding of the remaining 10 to 20 percent requires a restructuring of the entire program. Still, they conclude that such a translation can be worthwhile for large amounts of source text. They note also that maintainability of the translated code can be a problem; if the maintenance is done on Algol 60 code, then the manual part of the coding must be done repeatedly, and if it is done on the Ada code, then this code tends to be hard to read and understand.

Plaisted [12] gives an abstract language and methods for translating it into other languages. This language is imperative in nature, with arrays, pointers, and side-effects. However, the level of presentation is fairly abstract, without specifying in detail the semantics of pointers, for example. There are some corrections to this paper available from the author.

Another example of the language translation approach is the "filtering" approach [13]. In this paper, parsers of languages are expressed in pure Horn clause logic programming. The denotational semantics of languages are also expressed in pure Horn clause logic programming as functions from parse trees to semantics. Then one writes a logic program asserting that two programs in two different formalisms have corresponding semantics. By running this logic program, one obtains a translator from a program in one language to an equivalent program in another language. Because everything is expressed declaratively, the system is guaranteed to be correct, assuming all the Horn clauses are correctly specified. This system has been implemented and applied to some languages with quite complex specifications. This is really a platform for writing translators, in which some translators have been implemented.

One issue with the filtering approach is specifying the denotational semantics of high-level imperative programming languages with side effects, arrays, and pointers or structures. Another is making the logic program efficiently executable. There is no inherent guarantee that the logic program will be efficient or will even terminate (especially because it is in pure Prolog), so it must be written carefully to ensure its efficiency.

Brant and Roberts [14] discuss the SmaCC application, which has been used to create a wide variety of transformations ranging from simple refactorings to much larger ones, such as converting entire code bases between languages. They write, "The SmaCC Transformation Engine has been used by the presenters for transformations ranging from a custom Java refactoring that took 15 minutes to create, to a source code migration project that converted a 1.4 million line Delphi project into C$^{\#}$ without halting the development process."

Andrews *et al.* [15] write, "Automated source-to-source translation is an attractive vehicle for migrating legacy code out of a proprietary programming language. The techniques described here are motivated by the need to translate software comprising millions of lines of code, of which hundreds of thousands of lines are shared interfaces which contain many thousands of macros" (p. 281). "Source-to-source translation to a standard, portable, and widely-available language, such as C, has been shown to be an excellent means of making exotic languages available on many platforms" (p. 282). The authors discuss translations that preserve the structure of macros and files in the context of the Rosetta system that translates pTAL to C^{++}. This preservation makes the resulting code more readable and easier to maintain. They define fragments as portions of a program and translate fragments in one language to fragments in another.

Tansey *et al.* [16] discuss source-to-source transformation of JAVA code to JAVA with annotations. They write, "We demonstrate the effectiveness of our approach by automatically upgrading more than 80 K lines of the unit testing code of four open-source Java applications to use the latest version of the popular JUnit testing framework" (p. 295).

From these references, it should be clear that source-to-source translation from one language to another, or from one version of a language to another, is in many cases possible and profitable. The emphasis of this work is on preserving the structure of a program using mostly local information, and not on decreasing the running time. It may be necessary to restrict the translation to subsets of the original language to make this approach feasible, and in some cases, manual translation of parts of the program may be necessary. Of course, these references are only a small sample of the work that has been done in this area.

Also, Trudel *et al.* [17] discuss source-to-source translation of C to Eiffel, Verdoolaege *et al.* [18] discuss a source-to-source compiler of a sequential program for parallel execution on a modern GPU, and Song and Tilevich [19] discuss preserving non-functional aspects of languages in source-to-source translations,

Source-to-source translation has also been used for program optimization or development in a given language. For example, LLVM (the low level virtual machine [20]) can translate from any language supported by GCC (C, Ada, C^{++}, Fortran, Objective C, Objective C^{++}, or Java) to C, C^{++}, or the common intermediate language CIL (MSIL) [21]. LLVM includes extensive analysis, transformation, and code optimization facilities. One of the LLVM projects even uses a theorem prover to evaluate symbolic paths through a program. Also, ROSE [22], developed at Lawrence Livermore National Laboratory, can generate source-to-source analyzers and translators for multiple source languages including C, C^{++}, and Fortran. The intermediate representation (IR) used in ROSE is an abstract syntax tree, preserving all information from the source code, and is a possible candidate for an abstract language, as are other intermediate languages. ROSE includes tools for static analysis, sophisticated program optimization, arbitrary program transformation, domain-specific optimizations, complex loop optimizations, performance analysis, and cyber-security.

As another example of this approach, Standish *et al.* [23] discuss using interactive source-to-source transformations in the context of program improvement and program refinement.

Arsac [24] considers node-splitting transforms, which do not modify execution history and are asserted to be complete, as well as other transformations that modify the execution history. Used in an interactive system, these transformations are applied to program manipulation and development. For example, these rules sometimes permit recursive programs to be transformed to iterative ones.

Cameron and Ito [25] discuss metaprogramming for source-to-source program transformation, taking a high-level program and producing another, more efficient program in the same language. Their approach can be applied to transformation of programs or program fragments in the context of program development, but it also has other applications.

Source-to-source transformation can also be used to adapt a program to a different machine architecture. As an example, Kuck *et al.* [26] are concerned with FORTRAN compiler optimizations for several different types of high-speed architectures, but their approach can be applied to many high-level languages.

Lee *et al.* [27] discuss programming for general purpose graphics processing units (GPGPU's). They present a compiler framework for automatic source-to-source translation of standard OpenMP applications into CUDA-based GPGPU applications. OpenMP is convenient for programming, but CUDA from NVIDIA permits the use of graphics processing units with increased execution efficiency.

Basumallik *et al.* [28] "present compiler techniques for translating OpenMP shared-memory parallel applications into MPI message passing programs for execution on distributed memory systems" (p. 189).

There are other approaches besides source-to-source translation that permit programs in different languages to work together. As one example, the .NET system developed by Microsoft can combine programs in many languages (the CLI languages) and permit them to work together [29]. This system therefore has advantages in that programs written in one language can be used with programs written in another language. However, it requires current versions of all compilers for supported languages to be included in the system, increasing its complexity.

Molloy *et al.* [30] discuss testing equivalence of programs in different languages. Their approach is applicable for testing the code produced by a source-to-source translator to ensure that it is equivalent to the original program. Their system found bugs in their source-to-source translator, as well as bugs in the compilers of the original and target languages, and other types of defects. Concerning the general topic of source-to-source translation, they write, "Automated source-to-source translation is an attractive vehicle for migrating legacy software out of a proprietary programming language. Automated translation that preserves macros and source file structure is the most practical, economical, and reliable way to reduce and eventually eliminate dependence on a less desirable programming language while supporting huge bodies of legacy software…We have developed the Rosetta Translator, which implements this method. It has been used to translate software comprising millions of lines of code, of which hundreds of thousands of lines are shared interfaces" (p. 97).

5. Comments on Past Work

From the preceding references it is clear that many people are using source-to-source translation to advantage, so although the technique was rejected in the past for Ada, it is being widely used today. Sometimes it is difficult to know if the described systems are fully automatic or guaranteed correct, but the work is relevant either way. Even a system that requires some human interaction and some testing at the end still can be a significant help in software design. Of course, it is better if the system is automatic and guaranteed correct.

The translations defined by Albrecht *et al.* [10] used mainly local information, preserved the structure of the original program, and only applied to a subset of the original language. This gives us an indication of the kinds of translations that may be most helpful for avoiding recoding the same algorithm in many different languages.

At least the past work shows that it is possible to define large translatable subsets of languages and translate efficiently between them using mostly local information, even for fairly complex high-level languages. Thus the source-to-source approach is feasible.

Some of the past works (Wallis [9], Huijsman *et al.* [11]) rejected or questioned the source-to-source approach because it could not be made fully automatic or guaranteed correct, thus setting the bar too high. These researchers also required the source-to-source system to translate the entire program automatically, and to work for all programs. The fact that a small untranslatable portion of the program could require reprogramming the rest in some cases was also given as a reason to reject this approach.

Huijsman *et al.* [11] and Wallace [9] also mentioned the problem of maintaining the target program for source-to-source translation. However, if the structure of the program is largely preserved, as Albrecht *et al.* [10] and Andrews *et al.* [15] mention, then similar variable names can be used in the original and target programs, and comments can be inserted in corresponding places, making the target program easier to maintain without reference to the original program. Also, Andrews *et al.* [15] and Molloy *et al.* [30] feel that if the translation preserves macros and source file structure, then the target program can be maintained without reference to the original language.

Another problem with some of the previous work is that it was restricted to translations from other languages into Ada. Current languages may have greater applicability to source-to-source translation.

It was mentioned by Huijsman *et al.* [11] that the 20 percent of a program that needs to be coded by hand can sometimes cause the remaining 80 percent to need recoding, too; this is not necessarily a serious problem because such recoding may only be needed a small portion of the time, and even when it is, advances in programming languages such as abstract data types, objects, and encapsulation may restrict this problem further.

6. Possible Future Research Directions

Despite the successful use of source-to-source translation in the past, the full potential of this approach has not been realized. It is still frequently necessary to write the same programs over and over again in different languages. Examples of such programs include standard searching and sorting algorithms, algorithms for dictionaries, graph algorithms, maximum flow algorithms, number theory algorithms, encryption algorithms, fast Fourier transforms, string searching algorithms, and many, many more. Programs written by students in the past often become unusable because of a change of machines, languages, or versions of languages. This should not be so. Once a program is written, it should be possible to execute it at any later time despite changes in machines or languages. If the language or machine changes, then it should be possible to translate the program into a language that is still in use, possibly with a small amount of the code needing to be rewritten by hand.

Also, new software is frequently written from scratch even when portions of it may already exist in other languages or even in the same language. Source-to-source translation is not used much in typical programming projects. A wider use of this technique has the potential to make programming easier and the resulting programs more reliable.

How can the potential of source-to-source translation be more fully realized?

Based on the work surveyed above, it is possible to sketch approaches for making source-to-source translation a more integral part of the typical programming endeavor.

One significant feature of previous work is that it showed the value of only defining translations on well-behaved subsets of the original language, rather than the entire language. It was stated in Albrecht *et al.* [10], Wallis [9], and Huijsman *et al.* [11] that certain features of a language may be difficult or impossible to translate into another language; such features can be omitted from the well-behaved subset of the language. As an example, restrictions on actual parameters of procedures to prohibit the same actual parameter to be used for two formal parameters, may be helpful in some cases. A related technique is the translation of fragments of a program, rather than requiring the entire program to be translated.

Because of their simple syntax and semantics, abstract languages have the possibility to be easily translated into many other programming languages. These abstract languages could be at the level of the pseudo-code used for descriptions of algorithms found in typical algorithms textbooks; the essence of the algorithm is described, but inessential details and complexities of the particular language are omitted. This facilitates translation into other languages. Then programmers could be encouraged to code in such abstract languages, and translators from these languages into other languages could be written. Another possibility is to encourage programmers to code in well-behaved subsets of existing languages, such as were mentioned in Albrecht *et al.* [10] and Huijsman *et al.* [11]; then translators from these subsets into other languages could be written.

It's not often that an entire program to be written can be obtained from another language. It's more likely that portions, or fragments, of it exist in other languages. It would be convenient to be able to access these fragments and translate them into the implementation language to simplify the programming process.

For this purpose, it would be helpful to have source-to-source translators widely implemented and available to the community along with libraries of program fragments suitable for translation. With each such program fragment, a specification or text description of it could be kept, and means could be provided to search the library for programs satisfying a given specification or having a given text description.

Note that it may be helpful to have more than one translator out of a given language; one translator may apply to more features than another, which increases its range of applicability but also may increase its complexity and running time. Of course, it is easier to translate a feature into another language that already has a similar feature.

It would be good to integrate this approach with a testing tool, as was done by Molloy *et al.* [30], to execute corresponding procedures in the original and target languages on corresponding inputs and check if their behavior is the same. This approach may be able to identify in which fragment or procedure the error lies.

The main purpose of the source-to-source translation approach is not to translate legacy code into other languages, but to encourage new programs to be written in a way that facilitates source-to-source translation, and possibly identify fragments of existing code that are suitable for translation. Perhaps some legacy code can be made available to source-to-source translation, if it happens to be written in a suitable subset of its language, or can be modified to be so.

Differences from Current Practice

Source-to-source translation is currently used mostly to translate large complex application codes used by one organization internally or in a product, rather than smaller portions of programs used by many people, such as algorithms and modules with precise specifications. In contrast, what is being proposed here is to use source-to-source translation in different ways, namely,

1) Develop abstract languages or subsets of application languages and write source-to-source translators out of them into many other application languages.

2) Develop libraries of algorithms and modules in such abstract languages or subsets of application languages.

3) Use source-to-source not on large application programs but on algorithms and modules with precise specifications, used by many people.

It is also proposed to adopt a programming style in which portions of a program are obtained by translation from other languages and code is added to these by hand.

This approach would make more software building blocks available in more languages and would make these building blocks more reliable and accessible, thus aiding the process of software development.

7. Discussion

In general, a hand-coded program or a highly optimized library program from a library in the application language is likely to be more efficient than one obtained by source-to-source translation from another language. Such hand-coded programs can be optimized using guidance from design patterns. Also, if a suitable program is found in a library in the application language, then the best choice is just to take it from there. Then why would one use source-to-source translation? The answer is that a program translated from another language may be preferable to a hand-coded program or even a library program in the application language for reasons of productivity and reliability, even though such translation may degrade efficiency, and even efficiency may not be an issue in many cases.

7.1. Efficiency

First, efficiency is not always a major concern. Sometimes one just wants to get something running, perhaps for test purposes, perhaps for an application where running time is not critical. Then source-to-source translation is a good choice because it is easier than coding from scratch. If efficiency is a concern, then it is possible to hand optimize the translated program, perhaps optimizing only the inner loops, and thus get a considerable gain in efficiency with a relatively small effort. This may also produce a more reliable program than a hand-coded one if the original program in another language is reliable. A program hand-coded and optimized using guidance from design patterns is not guaranteed to be correct. Even if a program overall is time critical, and needs to be hand-coded, parts of it may not be time critical, and source-to-source translation may be acceptable for them.

Also, many algorithms require little more than arrays and possibly pointers or structures, features present in many languages, so translating from an abstract language might not significantly degrade efficiency. For programs using more language features, the situation could be different.

However, abstract languages can be developed with various combinations of features. Then there should be some abstract language sufficiently close to one's implementation language so that one could translate an abstract program from there instead of writing it from scratch in the implementation language, and not lose much efficiency.

7.2. Abstract Programs

It is proposed to write programs in abstract languages or subsets of languages having a simple syntax and semantics, and translate programs from there into other languages. The reason for this is that verification and analysis may be easier in such abstract languages than in complex application languages.

Also, such abstract languages are likely to last much longer than complex application languages. Abstract languages are much closer to mathematical notation than a typical programming language. A proof of correctness of a program in an abstract language is like a mathematical lemma that only needs to be done once, and never repeated. Complex application languages are harder to prove correctness in and tend to go out of use faster.

The number of abstract languages is likely to be small and they will tend to be closely related, so that it should be easy to translate between them. Therefore people all over the world can work on verifying libraries of abstract programs. Eventually more and more abstract programs can be formally verified and then translated into application languages with proofs also translated, increasing reliability. Verification is much harder with libraries written separately in many different languages.

Thus with the use of abstract languages, program reliability can continually increase with time, and verified programs can all be kept in one place and accessed by many people, rather than being kept in many different places that people don't know about and are hard to find.

If programs in other languages have been translated from an abstract language, then any corrections to these programs can be propagated back to the abstract language and thence to programs and libraries in other application languages, helping reliability. Thus abstract programs become reliable even if not verified, as fixes propagate back to them, and programs translated from such abstract languages also become reliable. Even if an abstract language does go out of use, its programs can be translated into another abstract language, with verifications, thus preserving reliability.

7.3. Reliability

Knowing that a program might be translated into other languages might make it cost effective to do a more thorough job of testing and verification, making verification more practical than it currently is.

If a program is known to be reliable or has been verified, one may want to translate it into other languages instead of trusting library programs in those languages. In fact, abstract programs can be verified, and the proof translated along with the program as in proof carrying code. Thus one need not trust the translator, but can just check the translated proof.

Translation may also help people move out of old and little used languages and thereby reduce the number of languages, thus increasing program reliability.

8. Libraries

Libraries in an application language serve a useful purpose and are heavily used today. However, there is a limit to what such libraries can do; otherwise we would not need to program at all, and could just get all our programs from the libraries. Source-to-source translation can be helpful when a desired program is not in a library in the application language.

8.1. Libraries May Not have Some Programs

Libraries in an application language generally only contain widely used programs already written in the language. This leaves out many programs, such as programs written or modified by users that they might want to have available in other languages. This may be the case, for example, when a language is no longer supported. Even when code is widely used, it may not end up in a library. It may be difficult for the compilers of a library to know how often various programs are used, or how reliable they are, to decide whether they should be in a library. Some code might be proprietary, and thus cannot be put in a library. It's also a significant effort to put code in the library because it ideally should be extensively tested. Therefore libraries may have a lot of inertia, so that in fast-moving areas of computer science it may take them a while to catch up. Also, putting too many programs in a library might make it unwieldy, or reduce reliability, so the maintainers have to be selective in what goes in; if there are too many similar programs, it can be confusing for the user to find what he or she wants. In addition, libraries can't be as responsive to the needs of local communities. For little used languages, there may not be any good libraries. Very simple programs such as binary search may not be in a library. Some libraries may be small, and lacking many programs.

One may want to use programs in another language that are not in the library, for whatever reason. This could include programs in libraries in other languages, programs that have not been extensively tested, or recently developed programs in another language. It might also include earlier versions of programs, or even obsolete programs, to compare them with current ones. Perhaps one does not have confidence in the programs in a library. In general, source-to-source translation permits rapid interaction between research being done in different languages, with programs developed by one group being immediately usable by the other group.

8.2. Libraries May Not Have Highly Optimized Programs

If one really wants highly optimized code, then even a library routine may not be optimized enough. Highly optimized code is generally more complex and therefore requires greater testing to insure reliability, so a library will not always have the most highly optimized code, in order to help ensure correctness. In addition, if library code is highly optimized, then it may be harder for the user to understand and verify and modify it for his or her own purposes. Furthermore, for some areas optimization and performance continually increase depending on recent theoretical developments, so a library cannot always keep up with the state of the art. Optimization techniques may also differ depending on preprocessing time, the size of the input data, whether the data is stored in main memory or in secondary storage, the nature of one's data, the architecture of one's machine, interaction with other parts of the program, and storage allocated, so one cannot generally have a program that is optimized for all applications. A library may not choose to have multiple versions of a program optimized for different requirements, so one cannot always rely on a library for extreme optimization requirements.

8.3. Libraries May Be Hard to Search

Even if a good library exists, it may be difficult to find what one is looking for there. One might not know the term that is used to describe a desired program in the library.

8.4. Source Code May Not Be Available

Programs from a library (if they have been compiled already) are harder to modify for one's own purposes than programs translated from another language, where one can make small changes with ease, especially if the programs are commented and the comments survive the translation in a helpful form. Also, a library program may be proprietary so that one may not have access to the source code to modify it to suit one's purposes. In such a case it may be preferable to translate a program from another language and modify it by hand.

8.5. Libraries Are Needed for Each Language

New languages and language revisions are being developed constantly, and all would benefit by having libraries of programs in the language, but such libraries take time and effort to develop and make reliable; in a new language, no such libraries may be available.

Source-to-source translation can help to develop libraries for new languages or versions of languages. One can translate from programs in several other languages, pick the best result, and modify it by hand if necessary. Translation also reduces the cost of implementing new languages, because the libraries can be constructed faster. Source-to-source translation can reduce the number of libraries one needs, and can reduce the number of programs in each one, because many programs can be obtained by translation from other libraries. This is especially true for languages that are very similar to each other. Having fewer libraries helps to increase reliability.

8.6. Reliability Issues for Libraries

If one simply programs a new library from scratch for each new language that is developed, and languages continue to go out of style, then there is no guarantee that program reliability will continue to increase with time. One has the same kind of problem as before, in that the same program is written over and over in different languages.

To increase reliability, one can compare the performance of a library program with a source-to-source program to test the correctness of both of them.

9. Conclusions

Though the source-to-source technique was rejected for porting code to Ada, it is widely used today. Many groups are still pursuing various forms of source-to-source translation with substantial success. However, the full potential of this method has not been realized yet.

It would be worthwhile to further develop and use this technology. For this to happen, more translators have to be written, abstract languages and translatable subsets of application languages need to be defined, and libraries of program fragments in abstract languages or subsets of application languages should be created and made widely accessible. Also, programmers have to be trained in this approach and convinced of its value. One cannot say for sure exactly how such a system or systems would be designed or used until more experience has been gained. At any rate, source-to-source translation systems have already been tested by many users and have been found to be practical and useful in some cases. The full impact of this technology cannot be evaluated, however, until it is more widely implemented.

This technology could lead to significant increases in productivity and reliability of software. The potential benefits include faster coding and more reliable software, though testing, debugging, and hand coding would still be necessary.

The approach outlined here could also lead to a change in the nature of programming. It could encourage people to think of programs as abstract objects, not tied to a particular language, rather than as objects in a particular programming language. This technique might also impact language design, in favoring languages that facilitate source-to-source translation into and out of other languages. Perhaps it would be an additional encouragement to standardization between languages, as well.

REFERENCES

[1] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen and T. Margaria, "Software Engineering and Formal Methods," *Communications of the ACM*, Vol. 51 No. 9, 2008, p. 55. doi:10.1145/1378727.1378742

[2] P. J. Denning and R. D. Riehle, "Is Software Engineering Engineering?" *Communications of the ACM*, Vol. 52, No. 3, 2009, p. 25. doi:10.1145/1467247.1467257

[3] E. Tenner, "Buggy Software: Achilles Heel of Big-Data-Powered Science?" *The Atlantic*, 2012. http://www.theatlantic.com/technology/archive/2012/12/buggy-software-achilles-heel-of-big-data-powered-science/265690/

[4] R. M. Bates, "Logic of Lemmings in Compiler Innovation," *Communications of the ACM*, Vol. 52, No. 5, 2009, p. 7. doi:10.1145/1506409.1506412

[5] E. Putrycz and A. W. Kark, "Connecting Legacy Code, Business Rules and Documentation," *Proceedings of the International Symposium on Rule Representation, Interchange and Reasoning on the Web* (*RuleML* '08), Vol. 5321, 2008, pp. 17-30.

[6] T. Ziadi, M. A. A. da Silva, L. M. Hillah and M. Ziane, "A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams," *Proceedings of the* 2011 16*th IEEE International Conference on Engineering of Complex Computer Systems* (*ICECCS* '11), Las Vegas, 27-29 April 2011, pp. 107-116.

[7] J. F. Cui and H. S. Chae, "Applying Agglomerative Hierarchical Clustering Algorithms to component Identification for Legacy Systems," *Information and Software Technology*, Vol. 53, No. 6, 2011, pp. 601-614. doi:10.1016/j.infsof.2011.01.006

[8] W. Gropp, E. Lusk and A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface," MIT Press, Cambridge, 1994.

[9] P. J. L. Wallis, "Automatic Language Conversion and Its Place in the Transition to Ada," *Proceedings of the* 1985 *annual ACM SIGAda international conference on Ada* (*SIGAda* '85), Vol. 5, No. 2, 1985, pp. 275-284.

[10] P. F. Albrecht, P. E. Garrison, S. L. Graham, R. H. Hyerle, P. Ip and B. Krieg-Brückner, "Source-to-Source Translation: Ada to Pascal and Pascal to Ada," *Proceedings of the ACM-SIGPLAN symposium on Ada Programming language* (*SIGPLAN* '80), Vol. 15, No. 11, 1980, pp. 183-193.

[11] R. D. Huijsman, J. van Katwijk, C. Pronk and W. J. Toetenel, "Translating Algol 60 Programs into Ada," *Ada Letters*, Vol. VII, No. 5, 1987, pp. 42-50. doi:10.1145/36077.36080

[12] D. Plaisted, "An Abstract Programming System," In: S. Shannon, Ed., *Leading Edge Computer Science Research*, Nova Science Publishers, New York, 2005, pp. 85-129.

[13] Gupta, *et al.*, "Semantics-Based Filtering: Logic Programming's Killer App?" *Lecture Notes in Computer Science*, Vol. 2257, 2002, pp. 82-100. doi:10.1007/3-540-45587-6_7

[14] J. Brant and D. Roberts, "The Smacc Transformation Engine: How to Convert Your Entire Code Base into a Different Programming Language," *Proceedings of the* 24*th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (*OOPSLA* '09), Orlando, 25-29 October 2009, pp. 809-810.

[15] K. Andrews, P. Del Vigna and M. Molloy, "Macro and File Structure Preservation in Source-to-Source Translation," *Software-Practice and Experience*, Vol. 26, No. 3, 1996, pp. 281-292. doi:10.1002/(SICI)1097-024X(199603)26:3%3C281::AID-SPE14%3E3.0.CO;2-0

[16] W. Tansey and E. Tilevich, "Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications," *Proceedings of the* 23*rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (*OOPSLA* '08), Vol. 43, No. 10, 2008, pp. 295-312.

[17] M. Trudel, C. A. Furia, M. Nordio, B. Meyer and M. Oriol, "C to O-O Translation: Beyond the Easy Stuff," *Proceedings of the* 2012 19*th Working Conference on Reverse Engineering* (*WCRE* '12), Kingston, 15-18 October 2012, pp. 19-28. doi:10.1109/WCRE.2012.12

[18] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado and F. Catthoor, "Polyhedral Parallel Code Generation for CUDA," *ACM Transactions on Architecture and Code Optimization*, Vol. 9, No. 4, 2013, 23 Pages. doi:10.1145/2400682.2400713

[19] M. Song and E. Tilevich, "Reusing Non-Functional Concerns across Languages," *Proceedings of the* 11*th Annual International Conference on Aspect-Oriented Software Development* (*AOSD* '12), Postdam, 25-30 March 2012, pp. 227-238. doi:10.1145/2162049.2162076

[20] [LLVM] http://llvm.org/

[21] [CIL] http://www.scriptol.com/programming/cil.php

[22] [ROSE] http://www.rosecompiler.org/

[23] T. A. Standish, D. F. Kibler and J. M. Neighbors, "Improving and Refining Programs by Program Manipulation," *Proceedings of the* 1976 *Annual Conference* (*ACM* '76),

Houston, 20-22 October 1976, pp. 509-516.

[24] J. J. Arsac, "Syntactic Source to Source Transforms and Program Manipulation," *Communications of the ACM*, Vol. 22, No. 1, 1979, pp. 43-54. doi:10.1145/359046.359057

[25] R. D. Cameron and M. R. Ito, "Grammar-Based Definition of Metaprogramming Systems," *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 1, 1984, pp. 20-54. doi:10.1145/357233.357235

[26] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proceedings of the* 8*th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (*POPL* '81), Williamsburg, 26-28 January 1981, pp. 207-218.

[27] S. Lee, S.-J. Min and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," *Proceedings of the* 14*th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PpoPP* '09), Vol. 44, No. 4, 2009, pp. 101-110.

[28] A. Basumallik and R. Eigenmann, "Towards Automatic Translation of OpenMP to MPI," *Proceedings of the* 19*th Annual International Conference on Supercomputing* (*ICS* '05), 2005, pp. 189-198.

[29] J. Prosise, "Programming Microsoft .NET," Microsoft Press, Redmond, 2002.

[30] M. Molloy, K. Andrews, J. Herren, D. Cutler and P. Del Vigna, "Automatic Interoperability Test Generation for Source-to-Source Translators," *Proceedings of the* 1998 *ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA* '98), Vol. 23, No. 2, 1998, pp. 93-101.