MoHCA-Java*: A Tool for C++ to Java Conversion Support

Scott Malabarba, Premkumar Devanbu and Aaron Stearns
Department of Computer Science
University of California, Davis
Davis, CA 95616
530-754-9469
malabarb, devanbu, stearns@cs.ucdavis.edu

March 8, 1999

ABSTRACT

As Java increases in popularity and maturity, many people find it desirable to convert legacy C++ or C programs to Java. Our hypothesis is that a tool which performs rigorous analysis on a C++ program, providing detailed output on the changes necessary, will make conversion a much more efficient and reliable process. MoHCA-Java is such a tool. It performs detailed analysis on a C++ abstract syntax tree; the parameters of the analysis can be specified and extended very quickly and easily using a rule-based language. We have found that MoHCA-Java is very useful for identifying and implementing source code changes, and that its extensibility is a very important factor, specially to adapt the tool to assist in the conversion of C++ code that makes extensive use of libraries to Java code that uses similar libraries.

INTRODUCTION

As Java becomes more mature and efficient, many people find it desirable to convert legacy C++ or C programs to Java. The complexity of this task varies greatly between programs. Sometimes the conversion is trivial; but sometimes high level program design must be reconsidered, and the entire program rewritten. Tools exist to automate the conversion process, generally relying on some form of text stream processing or partial parsing. This type of tool is effective for the easier cases of conversion. However, these tools have certain limitations—their capacity for analysis is limited, and any change which requires a design decision is not recognized or reported. Our hypothesis is that a tool which performs rigorous analysis on a C++ program, providing detailed output on the changes necessary, will make conversion a much more efficient and reliable process. MoHCA-Java is such a tool. It performs detailed analysis on a C++ abstract syntax tree (AST); the parameters of the analysis can be specified and extended very quickly and easily using a domain specific [8] rule-based language.

IMPLEMENTATION

MoHCA-Java is built as a layer of abstraction on top of the Gen++ code analysis language [4, 3]. Gen++ provides constructs for traversing and analyzing AST's generated by the front end of a C++ compiler. We have defined a simple rule-based language, dubbed $\mathcal{MJL}$ (MoHCA-Java Language), which is used to specify target patterns in the C++ program, and the action(s) to take when those patterns are found. The user simply supplies a file

*Moderately Helpful C++ Analyzer for Java containing analysis rules written in $\mathcal{MJL}$, which MoHCA-Java checks against the C++ AST. This allows for very rapid and flexible specification of MoHCA-Java functionality, including extension to cover custom C++ libraries or new features in Java or C++. $\mathcal{MJL}$ captures the semantics needed to perform basic pattern-matching on an AST, while dispensing with the complexity of a more powerful language like Gen++.

We designed the system to be easily extendable on a deeper level as well, in a modular style. The $\mathcal{MJL}$ parser, intermediate representation, and Gen++ analyzer program are distinct components connected by well-defined interfaces; it is possible to modify or replace one module without disturbing the others. For example, the $\mathcal{MJL}$ syntax and grammar are constantly evolving - this is done easily by editing the lex and yacc specifications and rebuilding the analyzer executable. Likewise, the intermediate representation of $\mathcal{MJL}$ or its interpretation can be optimized or refined by changing the C++ code used internally, or the Gen++ code for the analyzer, as appropriate. This simplifies the porting of MoHCA-Java to other C++ analyzer tools such as ASTLog [1]. Figure 1 provides a high-level overview of MoHCA-Java's design and function.

Rules take the general form *properties:conditions=action*. $\mathcal{MJL}$ contains constructs for nested conditions, boolean operations, slot traversal, and explicit invocation of rules. Rules can be assigned several optional properties, including name, category, difficulty and link to another rule. For the present, the "action" contains the message text to be printed upon satisfaction of the rule. Optionally, one may add a sed script which will perform the translation, if applicable. The language is sufficiently expressive to allow rules for a wide range of Java/C++ differences such as those listed in [2, 6].

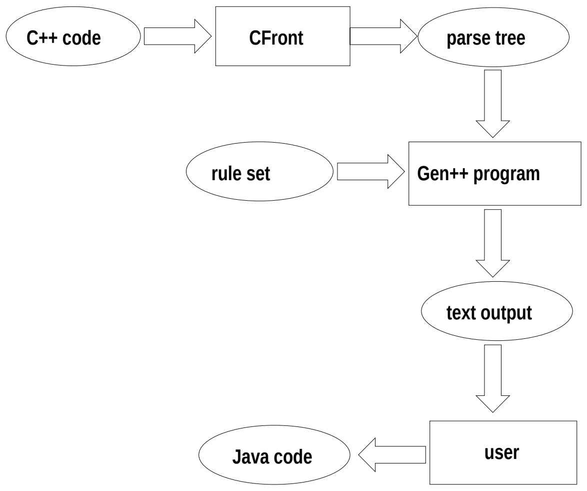

Figure 1: Overview of system.

Examples

Following are a few rule examples which demonstrate the basic syntax and semantics of $\mathcal{MJL}$.

- `(category library):`
  `(UserCall, printf):`
  `(msg "Use System.out.print instead of printf")(sed "sed script goes here...");`

  Look for any `UserCall` node with "`printf`" as its token string

- `(category design):(Goto):`
  `(msg "Goto statement not applicable in Java");`

  Look for an node of type `Goto`

- `(category library):`
  `(IdName)(or (IdName, cout)`
  `(IdName, cin)(IdName, cerr))`
  `:(msg "Use System instead of iostreams")`
  `(sed "sed script goes here..." ;`

  Look for any variable reference (`IdName`) with token string `cout`, `cin`, or `cerr`.

- `(category semantic):(If) (<ifcondition>`
  `not (or (Not)(Less)(Greater)`
  `(GreaterEq)(LessEq)(LogOr)(LogAnd))):`
  `"Non-boolean condition in if statement";`

  For all `If` statement nodes, check the `ifcondition` slot for a non-boolean expression.

ANALYSIS METHODS

We have cataloged differences between Java and C++ and separated them into categories and difficulty levels. The categories are syntax, semantic, library, and design. Syntactic differences involve simple changes in the syntax of an expression; for example, C++ classes contain public/private sections, while in Java each member is classified as public or private. Semantic differences are deeper, involving changes in the semantic definition of an expression. An example is the condition of an `if` statement - in Java this must be a boolean, unlike C++. Library changes are simply a switch from one function or global variable to another - say, from `printf` to `System.out.print`. Design issues require some redesign of the program, typically because of a feature which exists in one language but not the other - for instance, multiple inheritance or pointer usage. Difficulty level is a measure of the time and system knowledge required to enact a given change. We make the assumption that for most programs the bulk of the changes required will be low-difficulty syntax and library issues, with a few complex design issues on the other end of the scale. This categorization lends itself to efficient division of labor - the more numerous, time-consuming minor tasks could be assigned to a junior programmer, while a senior programmer handles more difficult issues requiring design decisions or specialized knowledge.

Output

Output consists of an HTML version of the source code analyzed - each line where rule(s) were matched is linked to a brief description of the change needed, and to a more detailed explanation in a separate file. Numerical data regarding number and type of changes needed is provided. A SED script file is also created, containing any SED commands corresponding to matched rules. This script is then run on the C++ source file to automate as much of the conversion as possible. Generally source files require some manual postprocessing before compiling in Java, as not all changes are completely automated.

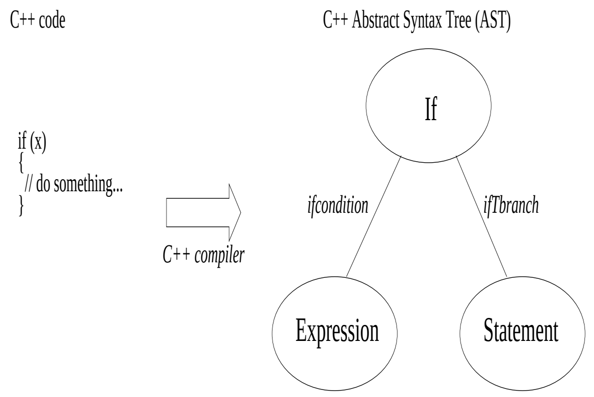

Figure 2: Analysis method.

RELATED WORK

Several similar tools exist, including C2J [7] and J2C++ [5]. C2J performs partial parsing of C++ files, generating Java source code according to a series of rules. C2J differs from MoHCA-Java in three significant areas:

- C2J generates complete Java code for all changes it is equipped to handle.
- MoHCA-Java permits full exploitation of all semantic information present in the AST.
- C2J translation rules are hard-coded and not easily customized or extended.

J2C++ provides Java wrappers for existing C++ classes using the Java native method interface. It does not actually translate the C++ code, only makes it accessible to Java programs. The advantage to this approach is that translation, with all of its inherent complications, is not necessary. However, for applications where source code conversion is required, J2C++ cannot assist.

In comparison with the others MoHCA-Java is more flexible and powerful, but automated conversion is not as seamless. Integration of translation capability into MoHCA-Java is an area of ongoing work. The ability to extend and use different rule sets is invaluable; creation of $\mathcal{MJL}$ rule sets for various existing C++ libraries is also a major focus of effort.

DISCUSSION

Currently, only partial translation is done, using SED scripts as provided by the author of the rule set. As any sort of text processing is very limited in semantic transformations, the user must code all other changes manually, using the output provided as a guide. MoHCA-Java would be even more useful if it could be extended to perform the translation directly on the AST wherever possible, leaving only those changes which are impossible to automate cleanly for the user to do. In this case a rule would specify both a pattern to search for in the C++ AST, and the transformation used to create the corresponding section of a Java AST. Gen++ does not have any mechanism for translation, so it will be necessary to either extend Gen++ or use another tool.

We have recently begun to test the system by using it to convert several programs of varying complexity both with and without its assistance, and comparing the time and expertise required for each task. Manually identifying necessary changes and locating them in code can be difficult, and we predict significant savings in time and energy. During development and initial testing we have found that MoHCA-Java is indeed very useful for identifying and implementing source code changes, and that its extensibility is a very important factor.

AVAILABILITY

MoHCA-Java requires an installation of Gen++, which is freely available for download [3]. MoHCA-Java itself is still under development and testing; we plan to make it available under a free source license on the GEN++ home page [3] by early Fall 1999.

Currently the basic $\mathcal{MJL}$ rule set covers most language differences between C++ and Java. Rule sets for various libraries such as the Unix system calls, string functions, etc. have also been written. Depending on available support, we also plan to port it to other source analysis tools [1].

References

[1] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings, First Usenix Conference on Domain-Specific Languages*, October 1997.

[2] Deitel and Deitel. *Java: How to Program.* Prentice Hall, 1998.

[3] P. Devanbu. The GEN++ page. `http://seclab.cs.ucdavis.edu/devanbu/genp`, 1998.

[4] P. Devanbu. Genoa - a customizable, front-end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology, (accepted, to appear)*, 1999.

[5] M. Hubbard and A. Schade. J2c++ developer tool for integrating c++ objects with java applets and applications. http://www.alphaworks.ibm.com/formula/J2C++.

[6] JavaSoft. The java language environment. Technical report, JavaSoft, 1996. http://www.javasoft.com/-docs/white/langenv/index.html.

[7] C. Laffra. C2j, a c++ to java translator. http://members.aol.com/laffra/c2j.html.

[8] J. C. Ramming. *Proceedings, First Usenix Conference on Domain-Specific Languages.* October 1997. (Edited).