Transforming Low-level Languages Using FermaT and WSL

DONI PRACNER AND ZORAN BUDIMAC, University of Novi Sad

There are many known problems in software maintanance, especially in cases where the available code is for whatever reason given only in low level executable versions. This paper presents a possible approach in understanding and improving such programs by translating it to an existing language *WSL* that enables the user to do formal, mathematically proven transformations of the resulting code. Such transformations can be done manually, but great improvements in the structure of the program can also be achieved by automatic scripts. Two prototype tools are presented that translate a subset of x86 assembly and a subset of Java bytecode, illustrated by examples showing the transformation process.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**] Distribution, Maintenance, and Enhancement

General Terms: Theory, Experimentation

Additional Key Words and Phrases: software evolution, FermaT, WSL, assembly, bytecode, transformations, translation

1. INTRODUCTION

One of the serious problems of modern software engineering is the perceived ageing of software. Although an application correctly written 20 years ago should still work as designed, maybe the underlying system is no longer available, making the application useless, or on the other hand maybe the user now needs a different result due to changes in the "real" world.

The problems of integrating legacy libraries, often available just as assembly code, into modern software/hardware systems can be tackled in different manners. One of the most efficient approaches, especially in the short term, is to encapsulate the functionality of reliable software[Sneed 2000]. Sometimes this is not really applicable – like in situations where new features need to be added to the system or, even worse, when there are bugs in the original software, in which cases it is necessary to understand and improve the original code.

The focus of this paper is on presenting two tools for working with low level code, that could help with understanding the logic behind the code and also potentially enable automatic restructuring of the code. One tool uses a subset of x86 assembly as its input, while the other one works with MicroJava bytecode. Both of the tools translate the programs into the high level language *WSL* that enables formally proven transformations on the source code, resulting in semantically equivalent code that should be much easier to understand.

The rest of the paper is organised as follows. Section 2 presents the existing transformation system that was used in this work. Section 3 shows the main steps and principles of the assembly translation and transformation process, as well as the tools created during this research. Following is an example

This work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support";
Author's address: Doni Pracner, Zoran Budimac, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: doni.pracner@dmi.rs zjb@dmi.rs

In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

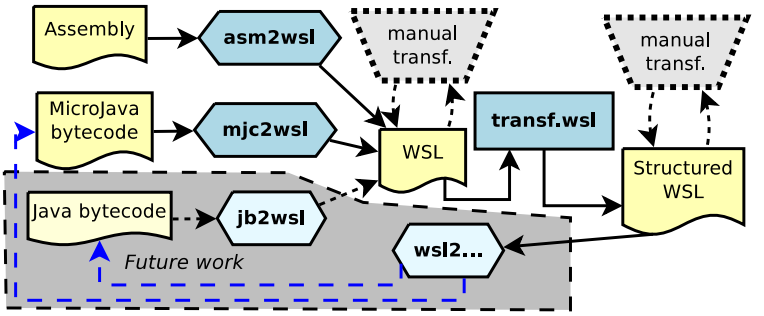

Fig. 1. Work-flow diagram of the tools – current and future

```
ACTION start:
start == code block 1 END
name1 == code block 2 END
name2 == code block 3 END
...
...
ENDACTIONS
```

Fig. 2. An action system

of the process, and some issues with the development. Section 4 introduces the bytecode tool, as well as illustrations of its work. Finally, conclusions, comparisons to related work and tools, as well as options for future work are given in Section 5.

2. WSL AND FERMAT

WSL (Wide Spectrum Language) is being developed by Martin Ward since 1989[Ward 1989]. A part of it is *MetaWSL* which gives the users constructs to write programs that will be able to transform code (internally represented as abstract syntax trees) using formal transformations. The current implementation is the *FermaT program transformation system*[Ward and Zedan 2005], and it is almost completely written in *MetaWSL*.

The main characteristics of the language is a strong mathematical core and the use of formal transformations, giving a reliable and provable system of improving software. The *wide spectrum* in the name means that there are constructs in the language that can be used for a wide spectrum of applications in development: from abstract specifications to low level program code.

The system was successfully used in many projects migrating legacy assembly code to maintainable C/COBOL code[Ward 1999; 2000; Ward et al. 2004]. There was also work on expanding the language to include support for concurrent programming[Younger et al. 1997] and object oriented programming[Chen et al. 2006], as well as incorporating a type system which improves the current state of transformations and provides a base for many future expansions[Ladkau 2009].

Action systems is a special structure in WSL which was specifically created to cope with unstructured jumps, which are very common in assembly code. It consists of a number of *actions* which can call each other, as shown in Figure 2. Once an action finishes it returns the control to the caller. Therefore an *action system* finishes when the start action finishes, or when a special, reserved action name “Z” is called which results in a momentary stop of the system.

Examples of action systems in use can be seen later in Figures 5 and 7.

3. ASSEMBLY TRANSFORMATION

Our transformation process consists of two basic steps: first we use our tool *Asm2wsl*[1][Pracner and Budimac 2011] to translate the assembly code to *WSL*, trying to capture all of the aspects of the original code without much effort to optimise at this step. Second, we use *Trans.wsl* (a script written in *WSL*) for automated transformations on the translated code. There is always a possibility to apply manual transformations, either before or after the automated transformations.

At its base, this approach is similar to those in the past that used WSL. But the difference is that our main goal in the process is to get a high level version of the original program that will represent all the aspects of its functioning. The approach using WSL presented in a number of papers [Ward 1999; 2004; Ward et al. 2004] creates additional files during the translation to WSL, which contain data about the variables and their mapping in the memory. These files are then used when the transformed and improved code is translated into (for example) *C* code and the appropriate pointer types are created.

Our approach should give a better understanding of the original code since we are looking at everything as high level structures, and it also enables us to run the translated programs directly in the WSL interpreter, without a need for an additional translator from WSL into another (semi) low level language. The downside is that assembler structures are often obscure, access data in different manners, and are therefore hard to understand and represent as high level, which limits the current version of the tools to a smaller subset of assembly code.

Asm2wsl is a tool that translates a subset of x86 assembly to *WSL*. Of course, being that there are many different flavours of assembly, and that it is very hard to automatically distinguish and adequately process them, a decision was made to focus on programs for a single type of assembler. The choice was the format first introduced in *Microsoft Macro Assembler (MASM)* that was consequently accepted by Borland's *Turbo Assembler (TASM)*[Borland International 1990], due to the familiarity with the later tool at our institution. To keep things simpler, at the moment it mostly presumes that we are working with an 80286 processor, the reason being that as they were developed, newer processors were mostly extended with more registers, options to work with bigger words and more specialised commands, which are not of great importance to the concepts we are translating.

The tool has been implemented in Java, making it platform independent and a good match to *FermaT*, which can also be run on a number of platforms. At its core, this is a line by line translator, with the focus on translating all aspects of the original code, without considering optimisation at this stage of the process. This generally results in programs that are much larger than the original assembly, but later on automatic transformations are able to reduce the size of the code. The same principle was successfully used in earlier translators which use WSL for transformations [Ward 2000; 2004; Ward et al. 2004].

Assembly commands work (more or less) directly with the processor. Being that high level languages do not do this, to capture all the aspects of these commands, we created a "virtual" processor. In it we have local variables to represent processor registers. Bits from the flag register are all defined as separate variables, which they practically are in the processor.

The processor can of course work with variables of different sizes, which introduces the need to work with them differently (the different scopes of values) and another problem – how to detect them? To handle this an additional overflow variable was introduced, and the translator tries to detect the size of the target variable from the original context. Based on the value the flag variables (in most cases overflow) are set like they would be in the real processor. An 80286 processor works with just two sizes: 8 and 16 bits, which is one of the reasons for presuming an older architecture in the study.

[1]the tool is available from the project's page http://perun.pmf.uns.ac.rs/pracner/transformations

The principle can then be tested on simpler examples and extended in the future as needed to bigger variable sizes.

In an *x86* processor *Low* and *High* parts of registers can be accessed independently (i.e. in 16 bits the lower and higher 8 bits, and analogue in bigger ones). Being that our goal were high level structures, we wanted to exclude direct memory operations, so these were implemented with additional operations that set the adequate parts of the register, at the same time preserving the potential side effects of the original code.

Labels in the original code are translated as *Action system* names (see Section 2). The whole system that we generate when we translate a "normal" assembly program is by nature *regular* (meaning that none of the calls ever return, that is, all of the actions just call other actions, and the system is finished with a CALL Z). Because of the special properties, these can be transformed easily into structured code.

Basic operations with arrays are also supported by the tool, with an automatic adjustment to the indexes, which is necessary since arrays start from 1 in *WSL*, and from 0 in assembly.

The processor's internal stack is implemented as a global list/array. The pop and push commands take and put elements on the start of this list directly. No additional checks (such as element size and compatibility, presence of elements on the stack) are performed, being that we presume to work with programs that worked correctly in their original form.

Macro structures are not translated at all at this stage of the development. On the other hand there are some special macro names that are recognised and translated directly into *WSL* code to enable input/output operations. For instance print_num x and print_str x are directly translated to PRINT(x). Similarly read_num and read_str are directly translated to *WSL* commands for reading numbers and strings, respectively.

The tool also has support for translating procedures from assembly. They are translated as nested Action Systems, so that local labels can be created, and it also enables us to return to the point of the original call once the procedure has finished its work. The translated programs worked from the interpreter without modifications. Transformations were also successful, despite the process resulting in action systems that are not *regular*. In the initial small tests, the procedures were simplified and then included in the main action system, as will be seen in an example in Section 3.

The second part of the process consists of a small program written is WSL that goes through the abstract syntax tree of the translated program, and tries to apply some of the available transformations on adequate nodes. All of the transformations implemented in WSL need to have procedures that will test if they can be applied to the given node, so this part of the code is relatively simple. For example, a transformation that unrolls a WHILE loop will (among other things) first check if the given node is in fact a WHILE statement.

Some of the important transformations are collapsing of the action systems into endless loops with exits in the middle and the subsequent transform of those into WHILE loops. At the same time constants are propagated through the code and various redundant parts are removed.

Example. This example is an illustration of how the transformation of procedures should work. *SumN* is a simple program with a call to a procedure that sums the top of the stack. The original assembly procedure is shown in Figure 3.

The whole assembly program is, more or less, just loading the data onto the stack and calling the procedure. As explained before (end of Section 3), the procedure will be translated into a nested action system, as shown in Figure 5.

Transformations are then applied to the obtained system – removing flags, collapsing action systems, and transforming the loops to WHILEs. The end result is the procedure transformed into a form shown in Figure 3, which is directly included into the main program inplace of its call.

```
sumn    proc
;take n from the top of the stack
;sum the next n top elements of the stack
        pop cx
        mov bx, 1
        mov ax, 0
        mov dx, 0
theloop:
        pop ax        ; get next from stack
        add dx, ax    ; array sum is in dx
        cmp bx,cx     ; is it the final one?
        je endp       ; skip to end if ti is
        inc bx
        jmp theloop
endp:
        push dx       ;result
        ret
sumn    endp
```

Fig. 3. Assembly version of the SumN procedure

```
cx := HEAD(stack);
stack := TAIL(stack);
ax := HEAD(stack);
stack := TAIL(stack);
WHILE bx <> cx DO
  dx := ax + dx;
  IF dx >= 65536 THEN dx := dx MOD 65536 FI;
  bx := bx + 1;
  ax := HEAD(stack);
  stack := TAIL(stack) OD;
stack := <dx> ++ stack
```

Fig. 4. SumN – transformed

```
ACTIONS A_S_start:
A_S_start ==
 ..... stack init etc .....
  stack := < n > ++ stack;
  CALL sumn;
  rez := HEAD(stack);
  stack := TAIL(stack);
  PRINT(rez);
  CALL end1
END
end1 ==
   CALL Z END
sumn ==
  ACTIONS dummysys:
  dummysys ==
    cx := HEAD(stack);
    stack := TAIL(stack);
    bx := 0;
    ax := 0;
    dx := 0;
    CALL theloop
  END
  theloop ==
    ax := HEAD(stack);
    stack := TAIL(stack);
  ..............
    bx := bx + 1;
    CALL theloop;
    CALL endp
  END
  endp ==
    stack := < dx > ++ stack;
    CALL Z END
  ENDACTIONS
END ENDACTIONS;
```

Fig. 5. SumN – translated to an *Action system*

The Main Issues for Further Development. Although the initial results on small programs proved to be successful, there are several inherent issues with this approach that, when combined, make future development of the assembly tool less likely.

First is the question of feasibility of obtaining only high level structures in the translation (without auxiliary files). Second is the problem of assembly not being very standardised, even the order of the operands is not a sure thing, macros are defined in different ways, there are huge architectural changes, quite often there are "hacks" in the code, input output is done through hard to detect structures, etc. The authors were of course aware of these problem from the start, but were willing to sacrifice a good piece of the input domain in favour of higher quality end products.

Another problem is the lack of a good assembly base to experiment on. Both of the previous points make the selection of code very hard, and all of this is worsened by the lack of practical experience with assembly at our institution, which, taking into consideration the legacy aspects of the work, will probably not change in the future. This lack of "feel" for assembly and how programs are typically written with it is another negative factor.

The combination of these factors, as well as the options for the second tool that will be presented bellow, have led to the decision to focus the development of the tool and examples for use mainly in Software Evolution and potentially other courses.

```
program P                          14: enter 0 1
{                                  17: const_0
  void main()                      18: store_0
  int i;                           19: load_0
  {                                20: const_5
    i = 0;                         21: jge 13 (=34)
    while (i < 5) {                24: load_0
      print(i);                    25: const_0
      i = i + 1;                   26: print
    }                              27: load_0
  }                                28: const_1
}                                  29: add
                                   30: store_0
                                   31: jmp -12 (=19)
                                   34: exit
                                   35: return
```

Fig. 6. MicroJava code and the translated bytecode

4. (MICRO)JAVA BYTECODE TRANSFORMATION

Java Bytecode is the language that is executed inside the standardised Java Virtual Machine[Lindholm et al. 2011]. In many ways it is similar to "classic" assembly languages, and therefore the reasons for translations and formal transformations apply here.

Most of the time bytecode is generated by a compiler from source code in the Java programming language, as it would be expected. But there are other languages and projects that try to use all the advantages of the standardised and very popular JVM that is available for most of the computer platforms that are in use today. For instance there are compilers for Python, Ruby, Pascal, C, Lisp, Scheme, PHP, JavaScript, and many other languages that produce Java Bytecode. The programming language Scala is compiled for either JVM or .NET machines.

The long term plan of this project is to build translators to and from WSL, that would allow both formal verification and transformation. While "regular" bytecode generated from Java can usually be decompiled quite successfully, this is not the case with code compiled from non-Java languages, or in cases where there was bytecode instruction injections for some particular purpose (such as persistance or additional security checks).

As a proof of concept the first step would be to work on a subset of the language. In this case the existing MicroJava specification was chosen.

MicroJava[2] was developed by Hanspeter Moessenboeck, for use in Compiler Construction courses with focus on the main features of a programming language without the distracting details[3]. For instance the only types are int and char primitives, arrays and basic class support. The concepts present in the MicroJava version of bytecode are very similar to "full" Java Bytecode, but simplified, with less instructions. It is also important to note that types are not explicitly encoded in MJ bytecode. An example of a program in this language and the code generated from it can be seen in Figure 6.

For the purposes of translating bytecode to WSL a new tool is being developed *mjc2wsl* (mjc – MicroJava Compiled). The basic concept are similar to asm2wsl – local variables are used to represent the registers, stack and other structures in the virtual machine. An example of translated bytecode can be seen in Figure 7.

The tool is currently in a closed prototype testing stage, but the plan is to publish it under an open source licence on the project's page[4].

[2]The name MicroJava may associate to "Java ME" (Micro Edition), but they are not related at all.
[3]Handouts for the course, available at http://ssw.jku.at/Misc/CC/
[4]http://perun.dmi.rs/pracner/transformations/

```
...              ACTIONS
                 .....
                 CALL a18 END
18: store_0      a18 ==
                 loc0 := HEAD(estack); estack := TAIL(estack);
                 CALL a19 END
19: load_0       a19 ==
                 estack := <loc0 > ++ estack;
                 CALL a20 END
20: const_5      a20 ==
                 estack := <5 > ++ estack;
                 CALL a21 END
21: jge 13 (=34) a21 ==
                 tempa := HEAD(estack); estack := TAIL(estack);
                 tempb := HEAD(estack); estack := TAIL(estack);
                 IF tempb >= tempa THEN CALL a34 FI;
                 CALL a24 END
24: ....         .....
```

Fig. 7. MicroJava Bytecode and its WSL translation

5. CONCLUSIONS AND FUTURE WORK

This paper presents two tools for translating low level languages to a high level language *WSL*, which provides commands and structures for formal, provable transformations of the resulting code.

The first tool, *asm2wsl*, works with a subset of the Turbo Assembler flavor of x86 assembly language. The code gets translated into WSL with set up variables and structures that emulate the operations of the processor including all of the side effects. The complete logic of the program is translated to high level structures, without any memory mappings, unlike previous approaches[Ward et al. 2004]. Needless to say, the resulting code tends to grow in size, but this is not important as a series of automatic transformations can reduce the length of the program while keeping the logic intact. Manual transformations can be applied at any point for improved end results.

The initial results on small test programs were good, but this approach has a lot of limitations – many structures in assembly are practically impossible to detect and translate into high level counterparts. The input output system is a big problem for translation. Finally the available applicable code base and the amount of work being done at our institution with assembly is just not big enough for good tests. Therefore the decision was reached to turn this tool into a mainly educational helper in software evolution and potentially some other courses.

The second tool, *mjc2wsl*, works with MicroJava bytecode, which is a subset of Java bytecode (used in Java Virtual Machines). The basic inner workings are similar to the first tool. The strict specification solves most of the problems described above.

Direct bytecode changes can be used for a number of applications. *DYPER* [Reiss 2008] and *J-RAF2* [Hulaas and Binder 2008] are monitoring resources through instrumentation. *ASM Framework* [Kuleshov 2007] is used by a number of tools, including Jython, JRuby, Eclipse and some of Oracle's persistance systems. *Soot* is used in a number of research tools, as well as students' courses [Lam et al. 2011].

The end goal of this project is the expansion of *mjc2wsl* to Java bytecode and to take advantage of *WSL* that has already shown good results in real life applications and it's formally provable transformations, that are not available in other tools, both for optimisation and verification.

The obvious future steps are improvements to MJ bytecode automatic transformations, as well as developing translators from *WSL* back to bytecode and testing the potential improvements, and tools for verifying the correctness of the transformations. Further steps would be the expansion of the transla-

tion and transformation process to the complete Java Virtual Machine specification including two way translation between bytecode and *WSL*.

WSL has no static type system, and therefore transformations can not check type consistency which is a possible source of errors. For "full" Java Bytecode this would be necessary. A *Wide Spectrum Type System* was developed by Matthias Ladkau in his PhD thesis[Ladkau 2009], but it is not yet fully integrated into FermaT. This system could be used to improve many of the transformations, which is one of the goals of this project.

Another path of development would be the adaptation of the tools and good examples for courses in software evolution, software engineering and others.

ACKNOWLEDGMENTS

This work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support".

REFERENCES

Borland International 1990. *Turbo Assembler 2.0 User's Guide*. Borland International.

Feng Chen, Hongji Yang, Bing Qiao, and William Cheng-Chung Chu. 2006. A Formal Model Driven Approach to Dependable Software Evolution. *Computer Software and Applications Conference, Annual International* 1 (2006), 205–214. DOI:http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/COMPSAC.2006.10

Jarle Hulaas and Walter Binder. 2008. Program transformations for light-weight cpu accounting and control in the java virtual machine. *Higher-Order and Symbolic Computation* 21, 1-2 (2008), 119–146.

Eugene Kuleshov. 2007. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development* (2007).

Matthias Ladkau. 2009. *A Wide Spectrum Type System for Transformation Theory*. Ph.D. Dissertation. De Montfort University, Leicester.

Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*. Galveston Island, TX.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2011. *The Java Virtual Machine Specification, Java SE 7 Edition*. Oracle Ameria Inc.

Doni Pracner and Zoran Budimac. 2011. Understanding Old Assembly Code Using WSL. In *Proc. of the 14th International Multiconference on Information Society (IS 2011)*, Vol. A. Ljubljana, Slovenia, 171–174.

Steven P Reiss. 2008. Controlled dynamic performance analysis. In *Proceedings of the 7th international workshop on Software and performance*. ACM, 43–54.

Harry Sneed. 2000. Encapsulation of legacy software: A technique for reusing legacy software components. *Annals of Software Engineering* 9 (2000), 293–313. Issue 1. http://dx.doi.org/10.1023/A:1018989111417 10.1023/A:1018989111417.

Martin Ward. 1989. *Proving Program Refinements and Transformations*. Ph.D. Dissertation. Oxford University.

Martin Ward. 1999. Assembler to C Migration using the FermaT Transformation System. In *IEEE International Conference on Software Maintenance (ICSM'99)*. IEEE Computer Society Press, 67–76.

Martin Ward. 2000. Reverse Engineering from Assembler to Formal Specifications via Program Transformations. In *7th Working Conference on Reverse Engineering, Brisbane, Queensland, Australia*. IEEE Computer Society.

Martin Ward. 2004. Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations. *Science of Computer Programming, Special Issue on Program Transformation* 52/1-3 (2004), 213–255. DOI:http://dx.doi.org/dx.doi.org/10.1016/j.scico.2004.03.007

Martin Ward and Hussein Zedan. 2005. METAWSL and Meta-Transformations in the FermaT Transformation System. In *IN COMPSAC*. IEEE Computer Society, 233–238.

Martin Ward, Hussein Zedan, and Tim Hardcastle. 2004. Legacy Assembler Reengineering and Migration. In *ICSM2004, The 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society.

E.J. Younger, K.H. Bennett, and Z. Luo. 1997. A Formal Transformation and Refinement Method for Concurrent Programs. *Software Maintenance, IEEE International Conference on* 0 (1997), 287. DOI:http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/ICSM.1997.624256