A Simple Mathematically Based Framework for Rule Extraction Using Wide Spectrum Language

Frederick V. Ramsey
Software Technology Research Laboratory
De Montfort University
Leicester LE1 9BH, UK
Frederick.Ramsey@students.dmu.ac.uk

James J. Alpigini
Pennsylvania State University
Penn State Great Valley School of Graduate Professional Studies
Malvern, PA 19355, USA
jja7@psu.edu

Abstract

Programs use rules to dictate or constrain specific decisions or actions. These rules have typically been tested, revised, and updated continuously; therefore, they represent a substantial and valuable business or intellectual asset. These rules often are not reused because the legacy program code is the only valid source for these rules, and extraction of the rules from the legacy code is thought to be too difficult. This problem is exacerbated when a re-engineering project potentially involves rule recovery from multiple programs in multiple languages. This paper reviews the uses of mathematically formal approaches to business rule recovery and extraction. The applications of provable transformations from different programming languages to Wide Spectrum Language (WSL) are reviewed, and a simple framework for two different rule extraction approaches using WSL is presented. An example of rule extraction using each approach is presented, and the requirements, advantages, and limitations of each approach are examined.

1. Introduction

Programs use rules to dictate or constrain specific decisions or actions. Rules are incorporated into these systems based on either the experiences or expectations of the organization or a subset of knowledgeable individuals, so that all users of these systems are guided by the same knowledge and constrained to identical behaviors. Rule revisions are typically made based on one of two factors: the organization's ever increasing understanding of its own successful practices, and the organization's response to a changing operational climate. These revisions reflect the real-time response to both internal and external changes, and also reflect the growing organizational knowledge and memory, and the associated 'state-of-the-company.' As these rules have typically been tested, revised, and updated continuously, they represent a substantial and valuable business or intellectual asset.

Unfortunately, these rule updates are all too often made only within the code. Due to this, no other accurate written records or documentation of these rules exists. When it becomes necessary to re-engineer these systems, these valuable rules are frequently not reused because the legacy program code is the only valid source of these rules, and their extraction from the legacy code is thought to be too difficult. The problem is further exacerbated when a re-engineering project potentially involves rule recovery from complex systems employing multiple programs in multiple languages.

Alpigini et al. [1] reviewed and classified the various rule-extraction techniques reported in the literature. From this review, several general problems in current rule extraction approaches emerge, especially when applied to larger heterogeneous systems involving both multiple rule analysts and languages. Firstly, there is substantial variation among researchers and practitioners regarding exactly what constitutes an extractable rule. While this is not a research failure *per se*, it does highlight the fundamental issue that various researchers and practitioners have different end-points, or expectations, regarding the rule extraction process. As a result, it may be difficult, or impossible, for one to use another's specific methodology or associated tool *if that methodology or tool embodies different expectations regarding what constitutes an extractable rule*. This lack of a clear standard regarding what constitutes an extractable rule makes the development, implementation, and assessment of a general rule extraction process, consistent across languages and platforms, extremely

difficult. For progress and success to be assessed, the target of the extraction process must be clearly defined, and agreed and accepted prior to initiating rule extraction project on a heterogeneous, multiple language system.

Secondly, many existing approaches and the related tools are language specific. Frequently, they are focused on unique and language-specific syntax, or on language-specific structures, such as pointers in the C language. Although these approaches were developed to address the specific problems or circumstances presented by a given language, such language specificity may compromise the use of many approaches in a heterogeneous language environment.

Finally, many rule extraction techniques are not mathematically formal or complete. The absence of mathematically formal elements in most current extraction techniques ultimately results in an underlying uncertainty regarding the completeness of the technique. An unintended omission of a critical rule or critical case would be certainly embarrassing, probably costly, and in the case of certain critical systems, possibly catastrophic. Most current techniques provide little basis for estimating the completeness of the extraction process and for assessing the possibility that a rule has been overlooked. Therefore, any final statement regarding the success of given rule extraction exercise can only be a reasoned opinion, instead of a demonstrable and supportable fact.

In response to these three shortcomings, a simple, mathematically based framework for rule extraction has been developed and is presented herein. Firstly, a general mathematical formality is introduced by describing a program in any language in terms of sets of language elements and structures. Secondly, the concept of an extractable rule definition function that specifies extractable rules in terms of language elements and structures is introduced. This rule definition function forces the *a priori* agreement regarding what constitutes a rule and assures the consistent search for and identification of rules using that definition. When an acceptable extractable rule definition function cannot be achieved, an alternative approach is identified based on the systematic exclusion of program structures that are not rules. Finally, to address the problem of language specificity, this general approach is implemented based on the translation and transformation of the original source program into an equivalent Wide Spectrum Language (WSL) program, and the subsequent rule analysis of that equivalent program. This general approach is demonstrated based on the conversion of a simple, rule-based, C language program example into an equivalent WSL program. The specific requirements, advantages, and limitations of this simple, mathematically based framework are examined, including both rule definition approaches. Finally, some specific directions for additional research are considered.

2. Formal Methods, Business Rule Extraction, and Reverse Engineering

The term *formal methods* refers to methods which have a sound basis in mathematics. To date, there has been little research specifically performed in applying formal methods to the specific problem of business rule extraction. However, the use of formal methods in related areas is well studied. Therefore, this section reviews the application of formal methods to the reverse engineering and re-engineering projects in general, and to the problem of code extraction in particular.

Liu et al. [2] reviewed of the use of formal methods in the re-engineering of computing systems. A five-away classification of formal methods was developed based on model based; logic based; algebraic; process algebra; and net based approaches. Within the context of this five-away classification, existing formal methods were reviewed with respect to their previous application in any reverse engineering domain. Consistent with their observation that formal methods are "both over-sold and under-used," only 4 of the 24 specific formal methods reviewed had been applied to reverse engineering.

Specific approaches to code extraction vary with both the target/source language and the extraction objective. Huang et al. [3] studied the business rule extraction process for COBOL programs. Using a formal definition of a business rule as a program segment that transforms a set of input variables into a set of output variables, program-slicing criteria were developed and used to identify code segments that contained business rules. Blazy and Facon [4] applied formal methods to the partial evaluation, i.e., the creation of specialized program segments based on specific input values, of Fortran 90 code. Villavicencio and Oliveria [5] combined both formal and semi-formal methods to reconstruct a formal specification from C language legacy code. The functional semantics of the resulting code slices were then expressed in the HASKELL programming language. Gannod and Cheng [6] applied both informal and formal methods to the reverse engineering of large systems written in the C language into formal specifications. Lanubile and Visaggio [7] presented a formal method for extracting reusable function code from poorly structured programs using the concept of the transform slice. Tan and Kow [8] implemented this transform slicing approach to identify the code elements that implement program functionality

in SQL programs. Zhou et al. [9] presented a language independent technique for formally assessing the critical behavior of a legacy system using inserted assertion points.

Although specific details vary with language and application, some common problems exist with regard to the application of these formal approaches to heterogeneous systems. Firstly, the suitability or applicability of a given formal method appears to be highly dependent on the program language or languages that are to be analyzed. Specific issues include the ability to map the specific program language semantics and syntax to the semantics of the chosen formal method; the *a priori* knowledge of the rule analyst regarding both the program language and the specific program code, including specific rule locations; and the desired outcome and/or anticipated use of the final product. Secondly, extraneous code is a major problem. Many researchers used informal techniques to preprocess the code prior to the application of formal methods [7, 10]. These informal methods were applied to both clarify and simplify the code. Finally, the participation of the human analyst is critical. Few reported approaches have been fully automated. Others require substantial domain knowledge with regard to specifying starting locations, variables, or specific input/output data. As business rule extraction is a subset of the larger domain of code extraction, these three issues pose a substantial challenge regarding the stated objective of using formal approaches for business rule extraction from heterogeneous systems.

Ward [11] describes a formal transformation-based approach to source code analysis and manipulation, including code extraction. This approach uses Wide Spectrum Language (WSL), a general programming language with a theoretical foundation and with semantics that are defined formally [12]. The general approach is the representation of a program, or a specific program element, as a function between the initial state of a given system prior to program execution and the final state of that system after program termination. By using a provable, mathematically sound transformation from a given programming language to WSL, a provably equivalent WSL program can be created. Then, by using provable program transformations within WSL, code can be moved, deleted, or merged, as appropriate, and equivalent higher level abstractions of the original program can be developed as needed. This general procedure of stepwise refinement using provable semantic-preserving changes for WSL transformations is described in Bull [13]. Using these equivalent WSL programs, the original program logic can be analyzed, specifications derived, or new code generated using a mathematically sound transformation from WSL to the new programming language. This approach has been successfully used to analyze and/or re-engineer programs where the original program code was in ADA, IBM Assembler, BASIC, CICS, COBOL, and Pascal [14 - 18].

With regard to rule extraction from heterogeneous systems, a WSL-based approach has numerous potential advantages. Firstly, using provable, mathematically sound transformations, programs in a variety of languages can be converted into WSL, thereby allowing its use with potentially any source language. Secondly, with its extraction methodologies for the WSL code could be developed and implemented with the certain knowledge that the underlying logical or mathematical objectives of those methodologies were being achieved. Thirdly, performing analyses in a single language, WSL, will allow the consistent execution of code analysis or rule extraction strategy regardless of the initial program language. Fourthly, different programs, written in different languages, different styles, and with different levels of extraneous code, e.g., error handling code, could be consistently abstracted using WSL. Finally, rules derived from different original source programs can be expressed easily in a common, consistent form.

3. Programs and Rules as Sets of Language Elements and Language Structures

Every programming language consists of a finite set of language elements. For the purpose of this analysis, elements are the atomic units of a language that have a single meaning, function, purpose, or otherwise represent a single value, entity, group, or class. In general, these language elements may include numerical values; variables; mathematical operators; logical operators; assignment operators; language-specific reserved or key words; language-specific punctuation, separators, delimiters, and terminators; and other language-specific commands or tokens necessary for the execution of the program code. For any given programming language, the syntactic composition required to create specific instances of these language elements are explicitly stated or defined. For example, mathematical operators may represented as a single symbol (+, -, *, or /), and a variable may be defined as a series of not more than 255 letters and numbers starting with a letter. Such instantiation syntax constitutes what most programmers know and practice as the language. In all languages, these syntactic requirements limit the total number of possible unique instances of these language elements.

For a program written in a given language, a subset of the available language elements is used to create the specific language structures that form that unique

program. These structures are constructed from language elements arranged by the programmer in a specific and unique sequence to accomplish an intended task. A basic example of a structure is the single line of code 'x = 1;' composed of the language elements of variable, assignment operator, number, and terminator. Complex structures might be composed of multiple structures, and in the extreme, an entire program could be considered a single structure. Regardless of the final complexity of the structures used, in all finite programs, the total number of language structures contained in that program is limited and therefore knowable.

If extractable business rule forms can be defined in terms of the set of language elements and structures, and all structures in a given program identified, then each identified structure can be assessed as to whether it is, or is not, a rule by whether it matches a previously specified rule structure. If all program structures are identified and can be compared in a two-value manner (yes/no) against the specified rule structures, then the rule analyst can assert, with mathematical certainty, that all rules of specific form(s) have been extracted from the program. If the identified program structures can only assessed in a three-value manner (yes/no/maybe) against possible rule structures, then the location and magnitude of any uncertainty regarding what may or may not be a rule can be quantified.

These concepts can be expressed symbolically using set builder notation. For any programming language, let E be the set of all elements of that language, and S be the set of all language structures that can be formed from these elements, subject to the syntactic constraints of the language. The set of all rules, R, that can be formed in that language can be defined as:

$$R = \{ x \mid x \in S \wedge f(x, E, S) \} \qquad (1)$$

where the function f is an extractable rule definition function that specifies the properties that a rule must have in terms of the language elements and structures. To be meaningful, this extractable rule definition function f must be a characteristic function.

In the given language of interest, any program, P, can be defined as a finite subset of all structures S, or $P \subseteq S$. Finally, the set of all extractable rules, R_E, contained in program P can be defined as:

$$R_E = \{ z \mid z \in P \wedge z \in R \} \qquad (2)$$

With these equations, the functional requirements of this approach are clear. First, a general rule definition must be developed and expressed as a function in terms of specific language elements and structures, as required by Equation (1). Second, all structures contained in a given program must be efficiently identified and elicited. If both requirements can be achieved, then the rule extraction process reduces to the intersection of these two sets, as described in Equation (2), and all rules within a given program can be identified with certainty. This inclusion approach provides a two-value solution to the rule extraction problem in that all structures within a given program either are, or are not, a rule.

If an acceptable extractable rule definition function cannot be achieved, then an alternative or exclusion approach can be formulated, using this same general framework. Using the previously defined sets P, a given program, and R_E, all extractable rules within that program, the structures in that program that are not rules can be described as the relative complement of these two sets, or:

$$P - R_E = \{ z \mid z \in P \wedge z \notin R \} \qquad (3)$$

Using this alternative approach, two requirements must be satisfied. First, and as before, all structures within the program must be efficiently identified and elicited. Second, those structures that are not rules, i.e., $z \notin R$ in Equation (3), must be identified. A rigorous definition of this excluded set is the subject of future research. In practice, rule analysts will probably know that certain language structures cannot be rules. A comment is one obvious example common to all languages. Other specific cases of non-rules will depend on the language, and the presumed attributes of rules. Thus, in the absence of a rigorous definition of what is a rule, this alternative, or exclusion, approach provides a three-value solution to the rule extraction problem: no, maybe, and yes. With this approach, some structures will be tagged with certainty and excluded from consideration as non-rules; the status of the remaining structures remains uncertain, as some will, and will not, contain rules.

Both approaches, given in Equations (2)-(3), have their place in practice, each with their associated advantages and disadvantages. If an explicit, precise, and acceptable definition of an extractable rule can be developed, such as the extractable rule definition function f in Equation (1), then the Equation (2) approach can be implemented and all rules can be identified with two-value certainty. If such an explicit, precise, and acceptable rule definition is unavailable, then the alternative approach, Equation (3), can be implemented. Although incomplete, this approach limits and identifies portions of code where rules may exist subject to the certainty and specificity of the criteria used to exclude non-rule structures.

4. Demonstration of Two Mathematically Based Rule Extraction Approaches

This section presents a demonstration of the inclusion and exclusion rule-extraction approaches presented in Equations (2) and (3) respectively. For each of these examples, a simple rule-based program, written in the C language, was developed, as presented in Table 1. This program accepts the user input of two values from the keyboard, manipulates these two input values to determine a test value, then assigns and displays a reply value based on the comparison of this test value against a specified criterion. By design, the code is very simple to provide the basis for clear and unambiguous examination.

Table 1. A Simple Rule-Based Program in C

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #define CRITERION 20

5   int main(void)
6   {
7       char reply_yes[10] = "Yes";
8       char reply_no[10] = "No";
9       char reply[10];
10      char buffer[80];
11      double input1, input2, test ;

12      // This is a demo program

13      printf("Enter first input: ");
14      input1 = atof(gets(buffer));

15      printf("Enter second input: ");
16      input2 = atof(gets(buffer));

17      test = input1 + input2 ;
18      test = test + 2 ;
19      test = test * 2 ;
20      test = test + input2 ;
21      test = test + 2 ;

22      if(test >= CRITERION)
23      {
24          strcpy(reply, reply_yes) ;
25      }
26      else
27      {
28          strcpy(reply, reply_no) ;
29      };

30      printf("The answer: %s \n", reply);

31      return(0);
32  }
```

The example C code contains 22 structures. For this demonstration example, a discussion and rigorous definition of what constitutes a structure is purposefully omitted. These structures are: three library reference structures (lines 1, 2, and 3); a global constant definition (line 4); a single program block structure (lines 5, 6, 31, and 32); five type definition structures (lines 7 through 11); one comment structure (line 12); two input capture structures (lines 14 and 16); three output display structures (lines 13, 15, and 30); five sequential mathematical assignments (lines 17 through 21); and one logical if structure (lines 22 through 29).

This simple rule-based program was translated into WSL. The resulting WSL program is presented in Table 2. This equivalent WSL program contains only seven structures. These structures are: one variable declaration structure (line 1, terminating on line 10); two output display structures (lines 2 and 4); two input capture structures (lines 3 and 5); one mathematical operation reflecting the transformation of the multi-line local procedure in the C code into a semantically equivalent single statement (line 6); and one logical if structure (lines 7 through 9) that incorporates related output display. This dramatic reduction in the total number of structures highlights one of the major potential advantages of this approach to rule extraction and identification, namely, that using WSL allows the substantial simplification or abstraction of a target program code. Whereas not all circumstances will result in the magnitude of code reduction observed here, the elimination of code that is superfluous to the logical functioning of the program, either through translation into the streamlined syntax of WSL or through the transformation of source code into semantically equivalent WSL code, greatly aids in the comprehension of the program and the corresponding identification of program components of interest.

Table 2. The Equivalent Program in WSL

```
1   VAR < input1 := 0.0, input2 := 0.0,
        test := 0.0 >:
2   PRINFLUSH("Enter first input: ");
3   input1 := @String_To_Num(@Read_Line(
```

```
        Standard_Input_Port));
4   PRINFLUSH("Enter second input: ");
5   input2 := @String_To_Num(@Read_Line(
        Standard_Input_Port));
6   test := input1 * 2 + input2 * 3 + 6 ;
7   IF test >= 20
8       THEN PRINT("Answer: Yes")
9       ELSE PRINT("Answer: No") FI;
10  ENDVAR
```

4.1 Implementation of Equation (2) – Inclusion Rule

For the first example, the implementation of Equation (2), an extractable rule definition function is required. Alpigini et al. [1] presents a four-part definition of an extractable business rule that is applicable here. One attribute of this definition is that an extractable business rule must be logically or mathematically operative. Using this rule attribute and considering the requirement that a rule be a structure, the extractable rule definition function for this example will be whether a given structure is logically or mathematically operative. Any structure that contains a logical or mathematical operator will be declared a rule; any structure that does not contains a logical or mathematical operator will be declared a non-rule. Upon applying this extractable rule definition function to the previously enumerated list of structures in the demonstration code, only two structures are found to fit the criteria of being mathematically or logically operative – the mathematical assignment of the test value (line 6) and the one logical if structure (lines 7 through 9). These rule structures are highlighted in Table 3. No other mathematical or logical operators exist. The other structures either contain no operators, or contain only assignment operators. For example, since the function calls in lines 3 and 5 are for string input and manipulation only, and are not related to the mathematically or logically manipulation of these strings, these structures are determined not to contain any rules. Therefore, subject to the continued acceptance of the extractable rule definition function used in this example, one can be mathematically certain, using the set requirements specified in Equation (2), that all rules contained in the example program have been identified.

Table 3. Rule Extraction Using Equation (2)

```
1   VAR < input1 := 0.0, input2 := 0.0,
        test := 0.0 >:
2   PRINFLUSH("Enter first input: ");
3   input1 := @String_To_Num(@Read_Line(
        Standard_Input_Port));
4   PRINFLUSH("Enter second input: ");
5   input2 := @String_To_Num(@Read_Line(
        Standard_Input_Port));
6   test := input1 * 2 + input2 * 3 + 6 ;
7   IF test >= 20
8       THEN PRINT("Answer: Yes")
9       ELSE PRINT("Answer: No") FI;
10  ENDVAR
```

Note: The extracted rules are shown in **bold**.

4.2 Implementation of Equation (3) – Exclusion Rule

For the second example, the implementation of the alternative approach of Equation (3), no extractable rule definition function is required. Instead, only a basic understanding of both extractable rules and the WSL language is needed. For this example, it is assumed with certainty that variable declarations, output/display structures, and simple input capture structures cannot contain rules. This allows the elimination of lines 1, 2, 3, 4, 5, and 10 as non-rules. These eliminated structures are depicted in italics in Table 4. Thus, only two structures remain that have not been eliminated – line 6 and lines 7 through 9. Subject to a continued acceptance of the criteria used to eliminate the non-rule structures, one can be certain that the eliminated structures contain no rules. However, given the approach and the associated lack of an adequate rule definition, uncertainty must remain with respect to whether the remaining code does or does not contain any rules.

Table 4. Rule Extraction Using Equation (3)

```
1   VAR< input1 := 0.0, input2 := 0.0,
        test := 0.0 >:
2   PRINFLUSH("Enter first input: ");
3   input1 := @String_To_Num(@Read_Line(
        Standard_Input_Port));
4   PRINFLUSH("Enter second input: ");
5   input2 := @String_To_Num(@Read_Line(
        Standard_Input_Port));
6   test := input1 * 2 + input2 * 3 + 6 ;
```

```
7    IF test >= 20
8          THEN PRINT("Answer: Yes")
9          ELSE PRINT("Answer: No") FI;
10   ENDVAR
```

Note: The eliminated structures are shown in *italics*.

5. Discussion

Although purposefully limited in scope, these two example cases highlight the requirements, similarities, advantages, and limitations of these two related approaches to rule extraction. Both the inclusion and exclusion approaches require the identification of all structures contained in the target code, including the consideration and resolution of complex structures that may be composed of multiple structures or structures within structures. Given the very limited size of this example, and given the well-known and well-understood syntax of the C language, identification of all structures in the original program language at the appropriate level of detail was a simple, straightforward matter. However, if the original target program is lengthy, or if the original program language is either poorly documented or little known to the rule analyst, or both, then the identification of all program structures in the original program could be a formidable mechanical and logical task. For example, given lengthy source code, the possible number of structures will increase exponentially with the number of lines of code making it difficult, or ultimately impossible, to identify all structures within a program. In contrast, conversion into WSL potentially allows the substantial simplification or abstraction of the target program code. Given the formal and well-defined semantics of WSL, both the translation from the original program language and subsequent internal WSL transformations can be performed using provable, mathematically sound transformation libraries. Thus, these tasks can be performed with reproducible and provable certainty.

Although a substantial computational task, Ward [15] demonstrated that translating source code into WSL, translating WSL into a target language, and performing the associated semantic transformations can be largely automated. The automation of the actual rule extraction from the resulting WSL program will depend on the selection of the rule definition function that is to be applied in the case of Equation (2), or the specification of which structure types are to be excluded as non-rules in the case of Equation (3). However, the translation and transformation of disparate source languages into WSL should largely standardize the type and scope of the structures being analyzed, and thus facilitate the automation of the actual rule extraction process.

The C source code example used in this paper contains only two simple rules – a five-line local procedure that sequentially manipulates two input values to determine a test value and a logical if structure that assigns a reply value based on the comparison of that test value against a specified criterion. This example code does not consider a frequent problem associated with real world code – that of distributed or delocalized rules. Depending on the initial system design, subsequent code maintenance practices, or on the nature of the rule itself, various components of a single rule may be distributed in different locations throughout the code. Incorporating WSL in the rule extraction process may provide an alternative solution by allowing provable semantic transformations that can bring disparate semantic elements into a more localized abstraction, depending on the nature of the rule delocalization. Ultimately the problem of identifying and extracting delocalized rules will likely require a combination of program slicing and transformation, described by Harman and Danicic [19] as "amorphous program slicing" and described by Ward [11] as "semantic slicing."

The two mathematically based approaches to rule extraction differ dramatically with regard to the necessity for and application of an extractable rule definition. The inclusion example, based on Equation (2), requires an explicit, precise extractable rule definition function, presented in terms of the WSL language. If such a function can be implemented for a given language, all rules can be extracted from a program code with absolute, mathematical certainty. Conversely, the exclusion example, based on Equation (3), requires no *a priori* definition of rules, only an ordered understanding of what is not a rule. Consistent with that compromise, the second example affords no mathematical certainty regarding whether all rules have been extracted.

This leads to an important issue common to both approaches, namely, *what constitutes an extractable rule.* Rules represent knowledge and must be semantically explicit relative to the applicable knowledge domain. Unfortunately, the only way to extract a rule from existing code is through the analysis and abstraction of the program syntax. To that end, a functional definition of an extractable rule must identify how the knowledge of a general rule can be represented by the specific syntax of the program or abstraction language. This example used a basic, descriptive definition of a rule focusing only on one attribute of a rule – that it be mathematically or logically operative. However, such a non-rigorous definition of a rule functionally compromises the stated

objective of introducing some measure of mathematical formality into the rule extraction process. Therefore, a sufficient extractable rule definition function must be developed in terms of general language elements and structures. These formal requirements for a rule definition function may best be addressed by the careful and thorough identification of the atomic forms that rules can take, and the algebra that can controls in the formation of complex rules using these atomic rules. With such atomic rules and rule algebra, an extractable rule definition function then can be developed in terms of general WSL structures, which can then be applied to any language and program that can be translated and transformed into WSL.

Finally, the value of such a simple framework must be addressed. Within the context of the multiple participants in a real-world rule extraction process, each with their own knowledge domain and limitations with respect to the domain of others, the mathematical simplicity of this framework is its true value. Each participant makes a unique contribution to either the syntactic and semantic understanding of his or her respective domain. Given that the mathematical talents of the team will vary, this simplified mathematical approach provides a flexible framework in which all can participate. Within this framework, techniques that are more rigorous can be applied as appropriate to a specific problem or issue, without compromising the participation of other domain experts in other areas. Using WSL as a common focal point allows information from the various domains to be gathered, transformed consistently, and analyzed on a common basis. As necessary, output for the analysis of the WSL program can be translated into a common presentation format or, as appropriate, into the specific formats to suit the various domain experts. Although simple, this framework is far from trivial in that it provides a sound and workable basis to achieve both the management goal of project team consensus and the technical goal of mathematical formality.

6. Conclusions

This paper has presented a simple, mathematically based framework for business rule extraction from heterogeneous programming languages using Wide Spectrum Language (WSL). WSL provides well-defined semantics that readily allows provable, mathematically based code manipulation. This framework is based on the assertions that programs are composed from language structures, and that extractable business rules can be functionally defined in terms of specific language structures and elements. The concept of an extractable rule definition function as a characteristic function was introduced. This led to the mathematical specification of the two fundamental approaches to rule identification: specify exactly what constitutes a rule and identify it; or eliminate what is certainly not a rule, and assume that what remains either is, or could be, a rule. A simple WSL language example of rule extraction using each approach was presented.

This mathematically based approach attempts to address the need for mathematical formality in rule extraction approaches and shows great promise. However, several significant issued must be addressed in order to implement this approach using real world code. Firstly, additional research must be conducted with regard to what constitute a rule, both logically and mathematically. A succinct definition of what a rule is and how it represents the domain knowledge that it represents is critical to the development of an effective and efficient rule definition function and the implementation of this proposed approach. Secondly, a general extractable rule definition function, as a characteristic function in terms of WSL elements, operators and structures, must be specified. These formal requirements for a rule definition function may best be addressed by the careful and thorough identification of the atomic forms that rules can take, and the algebra that can controls in the formation of complex rules using these atomic rules. Indeed, the entire issue of scalability of this framework to real-world circumstances rests with the ability to adequately define extractable rules in general terms relative to identifiable WSL structures. Finally, the set of structures that are not rules, i.e., $z \notin R$ from Equation (3), must be explored both logically and mathematically. A more rigorous definition of this excluded set could provide an improved mathematical basis for code reduction approaches derived from Equation (3).

7. References

[1] Alpigini, J.J., Neill, C.J., and Ramsey F.V., "Classification of Rule Extraction Techniques from Knowledge-Based Systems", *Proceeding of the IASTED International Conference Modeling and Simulation*, 2001, pp. 60-64.

[2] Liu, X., Yang, H., and Zedan, H., Formal Methods for the Re-Engineering of Computing Systems: A Comparison, *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, 1997, pp. 409-414.

[3] Huang, H., Tsai, W.T., Bhattacharya, S., Chen, X., Wang, Y., and Sun, J., "Business Rule Extraction Techniques for

COBOL Programs", *Journal of Software Maintenance: Research and Practice* , 1998, Vol. 10, No. 1, pp. 3-35.

[4] Blazy, S., and Facon, P., Application of Formal Methods to the Development of a Software Maintenance Tool, *Proceedings 12th IEEE International Conference Automated Software Engineering*, 1997, pp. 162-171.

[5] Villavicencio, G., and Oliveira, J.N., Reverse Program Calculation Supported by Code Slicing, *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 35-45.

[6] Gannod, G.C., and Cheng, B.H.C. A Formal Approach for Reverse Engineering: A Case Study, *Proceedings Sixth Working Conference on Reverse Engineering*, 1999, pp. 100–111.

[7] Lanubile, F., and Visaggio, G., "Extracting Reusable Functions by Flow Graph-Based Program Slicing", *IEEE Transactions on Software Engineering*, 1997, Vol. 23, pp. 246-259.

[8] Tan, H. B. K., and Kow, J. T., Extracting Code Fragment That Implements Functionality, *Journal of Software Maintenance and Evolution: Research and Practice* , 2001, Vol. 13, pp. 53-75.

[9] Zhou, S., Zedan, H., and Cau, A., A Framework For Analysing the Effect Of 'Change' in Legacy Code, *Proceedings IEEE International Conference on Software Maintenance (ICSM '99)*, 1999, pp. 411-420.

[10] Gannod, G.C., and Cheng, B.H.C., Using Informal and Formal Techniques for the Reverse Engineering of C Programs, *Proceedings Third Working Conference on Reverse Engineering*, 1996, pp. 249-258.

[11] Ward, M.P., The Formal Transformation Approach to Source Code Analysis and Manipulation, *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation,* 2001, pp. 185 –193.

[12] Ward, M.P., Proving Program Refinements and Transformations, Oxford University, DPhil Thesis, 1989.

[13] Bull, T., An Introduction to the WSL Program Transformer, *Proceedings Conference on Software Maintenance*, 1990, pp. 242-250.

[14] Bennett, K., Bull, T., and Yang, H., A Transformation System for Maintenance – Turning Theory into Practice, *Proceedings Conference on Software Maintenance*, 1992, pp. 146-155.

[15] Ward, M.P., Assembler to C Migration Using the FermaT Transformation System, *Proceedings IEEE International Conference on Software Maintenance (ICSM '99)*, 1999, pp. 67-76.

[16] Yang, H., and Bennett, K., Extension of a Transformation System for Maintenance: Dealing with Data-Intensive Programs, *Proceedings International Conference on Software Maintenance*, 1994, pp. 344–353.

[17] Yang, H., Liu, X., and Zedan, H., Abstraction: A Key Notion for Reverse Engineering in a System Reengineering Approach, *Journal of Software Maintenance: Research and Practice*, 2000, Vol. 12, pp. 197-228.

[18] Zedan, H., and Yang, H., A Sound and Practical Approach to the Re-Engineering of Time-Critical Systems, *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, 1998, pp. 220-223.

[19] Harman, M., and Danicic, S., Amorphous Program Slicing, *Proceedings Fifth International Workshop on Program Comprehension, (IWPC '97)*, 1997, pp. 70–79.