# SBVR-based Business Rule Creation for Legacy Programs using Variable Provenance

Pavan Kumar Chittimalli
TRDDC, TCS Research
Pune, India
pavan.chittimalli@tcs.com

Abhidip Bhattacharyya[*]
University of Colorado Boulder
Boulder, CO, USA
abhidip.bhattacharyya@colorado.edu

## ABSTRACT

Functionality of a software system that implements business operations can be captured using business processes and rules. To understand the 'as-is' processes and rules, the source-code is arguably the best source of knowledge. We present a novel method that combines program analysis and domain knowledge to create the descriptions for "*IT rules*", as a critical step towards extracting business rules automatically. We introduce and use the concept of '*variable provenance*' to propagate the domain descriptions into the source code to create Semantics of Business Vocabularies and Rules (SBVR) rules. In our experiments on sample, near-real-life systems, we could successfully annotate very large percentage (> 90%) of IT rules and enable to create SBVR rules. We present and describe the `ProgAnnotator` tool which is based on *variable provenance* and generates descriptions for IT rules in the source code and subsequently create SBVR rules automatically.

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Specification languages*;

## KEYWORDS

Business Rule Extraction, Static Program Analysis, Variable Provenance, Rule Annotation, SBVR

**ACM Reference Format:**
Pavan Kumar Chittimalli and Abhidip Bhattacharyya. 2019. SBVR-based Business Rule Creation for Legacy Programs using Variable Provenance. In *Proceedings of 12th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference), Pune, India, February 14–16, 2019 (ISEC'19),* 11 pages.
https://doi.org/10.1145/3299771.3299786

## 1 INTRODUCTION

Organizations today are facing numerous challenges with their IT systems for the ability to adapt to changing business environments, providing integrated multichannel and personalized user experience. The business structure, policies, and strategies change over a time-period in response to the market conditions and external regulatory reasons. Business system transformation is a process of adjusting the business activities and the IT systems that automate them to accommodate the above changes. In addition, these legacy systems may also require changes in order to respond to changing business environment, i.e., competition, superior business products and services, and preparing for merger and acquisition (M&A) activity. Such changes may include replacing, re-developing, or refactoring the existing information systems and often termed as IT transformation.

Business rules are extracted from the source code [17], [20], [9], [21], [12], an activity performed as part of IT system transformation, largely manually. The rules extracted from source code are mostly code snippets that implement the business rules; we refer to such code snippets as IT rules. The identified pieces of code may denote complex flows in the system [12], consisting of variables with cryptic names and perform specific operations defined by the business. The comprehension of such code pieces that denote IT rules helps in understanding the constraints and calculations in the IT system that implement business functionality. The knowledge of the constraints and the calculations for specific business contexts, if expressed using domain vocabulary, can provide vital inputs for IT system transformation or understanding the as-is systems. In industry, most of these activities are manual, time, and resource consuming, though aided by tools. The currently used manual practice makes this activity extremely tedious as it involves keeping track of several variables, database tables and the complex flows between them.

Business systems are usually data centric and transaction based. The flow of data in such systems is cryptic in nature. It starts from the database or screens and flows through a maze of software processes/procedures, getting transformed at multiple places, triggering creation of many other data-items during its journey, and finally residing in the database and/or the screen. To understand the provenance of the output data, it is critical to understand the above flow, thus making it important to understand the transformation of the data in every process/procedure [3], [5], [8]. This is extremely resource consuming, if done manually. Therefore, we have adapted the concept of *variable provenance* [11] in the context of a software process or procedure [14]. The *variable provenance* is informally defined as some text that describes the data held by a variable. If the provenances of all variables in a procedure are known, they can be combined using program analysis techniques to construct the conceptual relation between output and input data (represented by

[*]Author has done this work while he was working in TRDDC. He is currently pursuing his PhD at University of Colorado Boulder.

*ISEC'19, February 14–16, 2019, Pune, India*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6215-3/19/02...$15.00
https://doi.org/10.1145/3299771.3299786

corresponding variables). Such conceptual relations are the basis for identifying and constructing business rules.

The Program Annotator tool is an Eclipse Plug-in, which generates descriptions for variables and statements of programs automatically from the available descriptions of entities and attributes of the persistent data like database schema, file records, and *i/o* records. We use static data-flow and control-flow analysis techniques to propagate descriptions of multiple variables in the source code. The tool computes the descriptions for conditional expressions by using the operational rules associated with mathematical and logic operators in the conditional expressions. The tool displays the descriptions as annotations when the user hovers the mouse on statements/variables of interest and it also enables users to edit the descriptions. Once the user validates approves the generated descriptions, we use technique [7] to generate the SBVR rules and vocabularies. These rules can be further used for verification and validation [10].

The main contributions of this paper are:

- Adapting the novel concept of *variable provenance* that accurately computes the descriptions of variables, expressions and statements.
- Creates the SBVR rules for the generated annotations.
- A prototype tool that implements the concept of *variable provenance*, and integrates with Eclipse as Plug-in.
- A set of empirical studies that show the effectiveness of the technique.

The paper is organized as follows: The Sect. 2 describes the motivating example for creating IT rules and Business rules using *variable provenance* technique. The Sect. 3 describes the detailed approach of *variable provenance*, statement annotation, combining them to create IT rules, and creating SBVR rules. The Sect. 4 describes the `ProgAnnotator` tool implementing the approach and experimental studies, The Sect. 5 describes the related work, and Sect. 6 describes the conclusions and challenges involved.

## 2 MOTIVATING EXAMPLE

In this section, we illustrate the concept of *variable provenance* and its application towards creating the IT rules using sample code fragment shown in Fig. 1. The code fragment is part of a money transfer module in a bank which accepts source account, target account, amount of transaction and updates the source and target accounts after the successful transfer of funds. The *i/o* for the code fragment is read/written using database (which is not part of the code fragment shown).

The variables `SRC-BALANCE`, `SRC-AC-ID`, `SRC-TYPE` get their values from `ACCOUNT TABLE`, and `AMOUNT-TRNDATE` gets its value from `TRANSACTION TABLE`. These variables are used in various computations and conditional expressions to calculate the output, depicted by `NEWSBALANCE` and `NEW-TBALANCE`. The descriptions available in the database denote annotations for the *i/o* (seed) variables. After the propagation of annotations from the seed variables to other variables in the code fragment, the annotations for few, sample variables as well as sample statements are shown in Fig. 1 as a tool tip (blob).

The annotation for variable `NEW-TBALANCE` on line `017700` is inferred from that of the variable `TGT-BALANCE`, as per the propagation rule of the addition operator. Thus the annotation "*Current balance of the account*" is assigned to variable `NEW-TBALANCE` and the description for the statement on that line "*Current balance in the account* `is computed as` *Amount* `is added to` *Current balance in the account*" is generated. The custom defined generation of the descriptions are better interpreted and illustrated than fixed generation format. For example, consider the above statement annotation generated as "*Current balance in the account* `is computed` `by adding` *Amount* `to` *Current balance in the account*". Both these generated descriptions convey the same meaning. The template based generation can assist such custom defined annotation generation to reflect the user interpretation (this is explained further in Sect. 3). In line `016100`, the annotation of `SRC-ID` is copied into `LACCOUNT-ID` as per the propagation rule of the assignment operator (explained in Algo. 3.3).

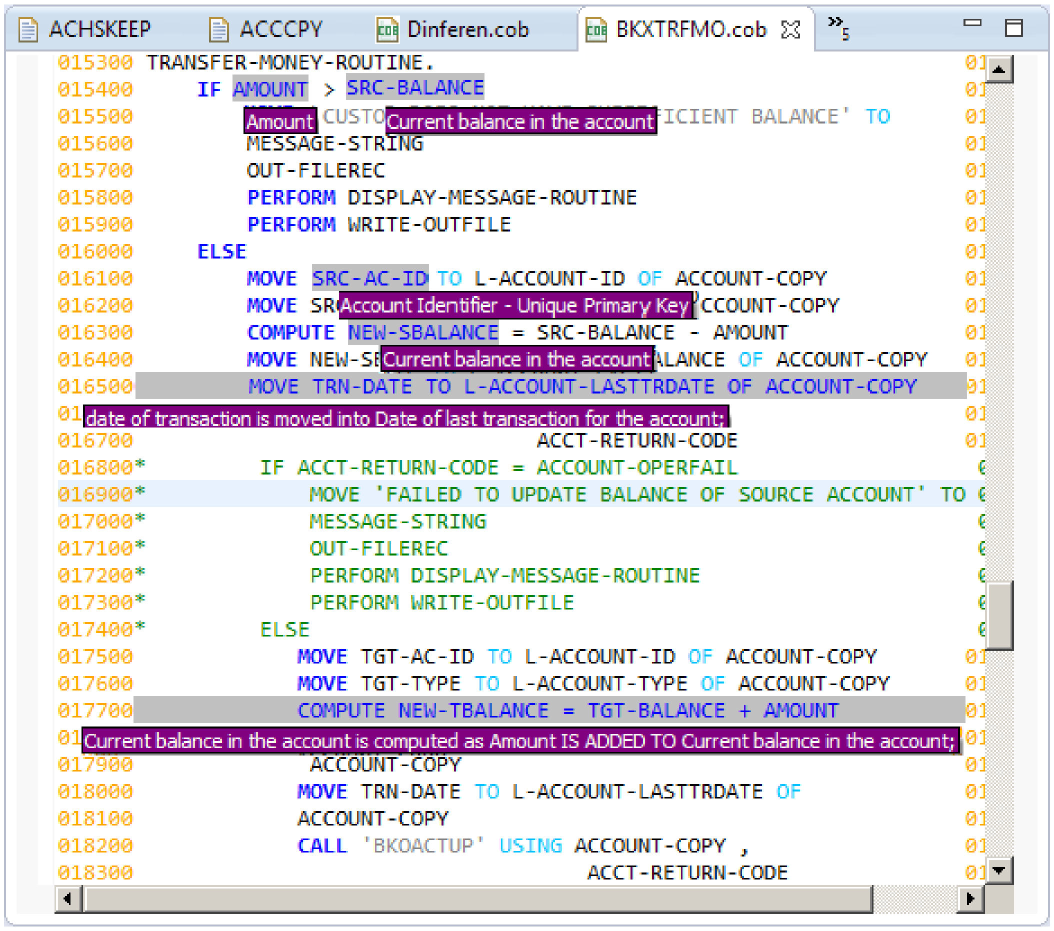
The descriptions for conditional expressions can be computed using the *variable provenance*. For an example at line `015400`, the generated description for the condition expression is "`If` *Amount of Transaction* `is greater than` *Current balance of the account* `then`". Similarly, the annotation for line `016300`, "`Decrease` *Current Balance of the account* `by` *Amount of Transaction*" is generated.

Most of the statements in the program get the annotations at the end of annotation propagation. It is possible that some statements (that do not involve any direct or indirect domain variables used in them) might not get annotations. For example, a loop counter is unlikely to have annotation and does not result in any descriptions involving that variable. Usually, all conditional and computational statements in the source code are annotated with descriptions. The developer and designer can understand the statements better in the context of the descriptions of the variables and the statements, enabling him to create a mental model of the business logic.

To document the business rules that are implemented in the source code, the analyst looks into the annotations of conditions, computations and variables, along with program analysis techniques and creates a description of the IT rule using the acquired mental model of the business logic. For a IT rule that is mapped using source lines between `015400` to `015900` (which is shown below) is created by the analyst can look like:

```
015400      IF AMOUNT > SRC-BALANCE
                  01540009
015500          MOVE  CUSTOMER DOES NOT HAVE SUFFICIENT
     BALANCE' TO  01550009
015600          MESSAGE-STRING
                  01560009
016000      ELSE
                  01600009
016300          COMPUTE~NEW-SBALANCE = SRC-BALANCE -
     AMOUNT             01630009
017700          COMPUTE~NEW-TBALANCE = TGT-BALANCE +
     AMOUNT             01770009
```

The identified IT rule can be annotated with the statement annotations as shown Tab. 1. Such IT rules consisting of domain terms are easier to comprehend by analysts for taking further decisions regarding IT transformations. These rules can be more useful for automatic transformations if specified in a machine interpretable formats such as Semantics of Business Vocabulary and



```
015300 TRANSFER-MONEY-ROUTINE.                                           01
015400     IF AMOUNT > SRC-BALANCE                                       01
015500         Amount CUSTOCurrent balance in the accountICIENT BALANCE' TO  01
015600         MESSAGE-STRING                                            01
015700         OUT-FILEREC                                               01
015800         PERFORM DISPLAY-MESSAGE-ROUTINE                           01
015900         PERFORM WRITE-OUTFILE                                     01
016000     ELSE                                                          01
016100         MOVE SRC-AC-ID TO L-ACCOUNT-ID OF ACCOUNT-COPY            01
016200         MOVE SRAccount Identifier - Unique Primary Key'CCOUNT-COPY 01
016300         COMPUTE NEW-SBALANCE = SRC-BALANCE - AMOUNT               01
016400         MOVE NEW-SCurrent balance in the accountLANCE OF ACCOUNT-COPY 01
016500         MOVE TRN-DATE TO L-ACCOUNT-LASTTRDATE OF ACCOUNT-COPY     01
01date of transaction is moved into Date of last transaction for the account; 01
016700                                  ACCT-RETURN-CODE                 01
016800*          IF ACCT-RETURN-CODE = ACCOUNT-OPERFAIL                  0
016900*              MOVE 'FAILED TO UPDATE BALANCE OF SOURCE ACCOUNT' TO 0
017000*              MESSAGE-STRING                                      0
017100*              OUT-FILEREC                                         0
017200*              PERFORM DISPLAY-MESSAGE-ROUTINE                     0
017300*              PERFORM WRITE-OUTFILE                               0
017400*          ELSE                                                    0
017500             MOVE TGT-AC-ID TO L-ACCOUNT-ID OF ACCOUNT-COPY        01
017600             MOVE TGT-TYPE TO L-ACCOUNT-TYPE OF ACCOUNT-COPY       01
017700             COMPUTE NEW-TBALANCE = TGT-BALANCE + AMOUNT           01
01Current balance in the account is computed as Amount IS ADDED TO Current balance in the account; 01
017900              ACCOUNT-COPY                                         01
018000             MOVE TRN-DATE TO L-ACCOUNT-LASTTRDATE OF              01
018100             ACCOUNT-COPY                                          01
018200             CALL 'BKOACTUP' USING ACCOUNT-COPY ,                  01
018300                                  ACCT-RETURN-CODE                 01
```

**Figure 1: Sample COBOL Money Transfer code segment.**

**Table 1: Sample annotated IT rules shown in the example rules.**

| Rule Id | Created Annotation |
|---|---|
| Rule-1 | If *Amount of Transaction* is greater than *Current balance of the account* then CUSTOMER DOES NOT HAVE SUFFICIENT BALANCE |
| Rule-2 | If *Amount of Transaction* is not greater than *Current balance of the account* then<br>Decrease *Source account balance of the account* by *Amount of Transaction*<br>and<br>increase *Target account balance of the account* by *Amount of Transaction* |

**Table 2: SBVR rule for annotated IT rules shown Tab. 1.**

| Rule Id | SBVR Annotated Rules |
|---|---|
| Rule-1 | it is necessary that if amount of transaction *is greater than* current balance of account then customer *is not having* sufficient balance |
| Rule-2 | it is necessary that if amount of transaction *is less than* current balance of account then source account *is decreased with* amount of transaction and target account *is increased with* amount of transaction |

Rules (SBVR) [1]. The corresponding SBVR rules created automatically by using our technique [7] is shown in Tab. 2.

## 3 APPROACH

In this section, we describe our approach of annotating IT rules using `Variable Provenance` technique [11] and translating these IT-Rules into SBVR based business rules using techniques defined in [7]. Our approach consists of following steps:

(1) Build the `Data Descriptions Table (DDT)` consisting of descriptions of database tables, file records, constants used in the system using the schematic descriptions.
(2) Identify the *i/o* points in the system which uses the databases/files and build the initial `Data Dictionary (DD)`. The DDT descriptions flow into the system at *i/o* points.
(3) We identify the variable provenances by applying data flow analysis. We identify equivalent sets of variables using the reaching definitions [3] and the annotation propagation rules of operators- a heuristic driven approach. The variables in one equivalent set have values that are of the same kind, either it is *amount*, or *interest rate*, or *tax-amount* etc.
(4) Compute annotations of statements by using annotations of individual variables and expressions. These are subsequently used for constructing the annotations for IT rules.

The detailed approach is shown in Fig. 2 and explained in the later part of this section.

### 3.1 Data Description Table (DDT)

The *i/o* variables (seeds) of the program are the variables which are external from sources. For example, the database table schema, file records, constants used in the source code carry external descriptions in a typical business application. In this step, the Intermediate Representation (IR) of source files generated by program analysis workbench PRISM [2] and extract descriptions for seed variables.

(1) **Table Data Descriptions:** The column names of a *table* and the corresponding descriptions are taken from the database schema.
(2) **File Data Descriptions:** In business applications, the data files are defined using custom defined structure. The file structure has definitive meaning to each of the column that is being stored in them. This information along with descriptions are captured in DDT.
(3) **Constants Descriptions:** The program may contain constants which hold some values and are used to assign value to the variables defined in the program. All such constants are typically stored in tables using maps.

The DDT is formally defined as: DDT = $\{< f, v, d >\}$, where $f$ is the database table or file type, $v$ is a record-field/constant/database-column, $d$ is the description associated with the $v$.

The DDT consists of the database variables (used in the database statements like CREATE/UPDATE/DELETE/INSERT along with database schema descriptions. It also contains the file handling statements like READ, WRITE uses file records. The schema definitions of those file records having file-record-fields descriptions are stored in the description table. Finally the constants that are used in the program also be stored in *DDT* using the *constant map* descriptions.

An example entry in the DDT for a record variable in a FILE as: < IN-FILE, SRC-AC-ID, "Account Identifier Unique Primary Key" >. Similarly for a *constant* 'S', the DDT entry will look as: < NULL, 'S', "SAVINGS" >.

### 3.2 Data Dictionary

The provenance of the *variable* is stored in data store, we call it *Data Dictionary* (DD). The DD is formally defined as: DD = $\{< v, d, eq >\}$, where $v$ is the variable, $d$ is the variable description (annotation), and $eq$ is number of the equivalence class.

Variables are stored in the data dictionary along with their descriptions and equivalence class number. We have shown the structure of data dictionary above. The DDT is computed earlier would be used for initializing the DD as shown in Algo. 3.1. Algorithm 3.1 initializes the entries of DD with -1 as equivalence class for all the DDT entries.

In the next section, we describe the operational rules associated with mathematical and logic operators; the heuristics developed and how data dictionary actually gets updated.

### 3.3 Variable Provenance Computation

In this section, we describe the algorithms `UpdateDD()` and `ComputeVariableProvenance(P)` to propagate and update data dictionary the annotations.

The ***domain variables*** are input and output points of a business system. The descriptions of these domain variables are the starting point for computing *variable provenance* given in Algo. 3.2. Usually, the descriptions of such variables are available as meta-data in the relational databases or file structures. In the Step (1), we create description tables for the database attributes, the file records, and the constants using the meta data available (`CreateDescriptionTab()`). The user-defined descriptions can be given to those variables for which the descriptions are not available externally. In Step (2), we create initial data dictionary (`InitializeDD()`) using the descriptions table created in Step (1). In DD, all the variables belonging to the same equivalence class refer and deal with similar or same data. For an example, we have equivalence classes such as ACCOUNT BALANCE, ACCOUNT TYPE for variables shown in Figure 1. In Step (3), we normalize the data dictionary using `NormalizeDD(DDT)`. The normalization of DD is performed to eliminate the duplicate entries.

In Steps (4-6), we compute *domain variables* and segregate them into input and output domain variables. In Steps (7-8), we perform backward data slice computation (`ComputeBackwardDataSlice()`) for each output domain variable. We use PRISM [2] data-flow analysis framework to compute slices. In Step (9-10), we take each statement from the backward data slice and perform updation of the Data Dictionary (`UpdateDD()`). This detailed algorithm is given in Algo. 3.3.

In Step (11-14), we perform forward data slice computation (ComputeForwardDataSlice()). Similar to Steps(7-10), we perform the updation of DD for statements in the forward data slice. Based on the heuristics developed from the annotation rules of operators, descriptions of variables from initial data dictionary flows through the program slice and new equivalence classes get created and merged. For an example, consider the statement in Fig. 1 at line number `016300 COMPUTE NEW-BALANCE = SRC-BALANCE -`

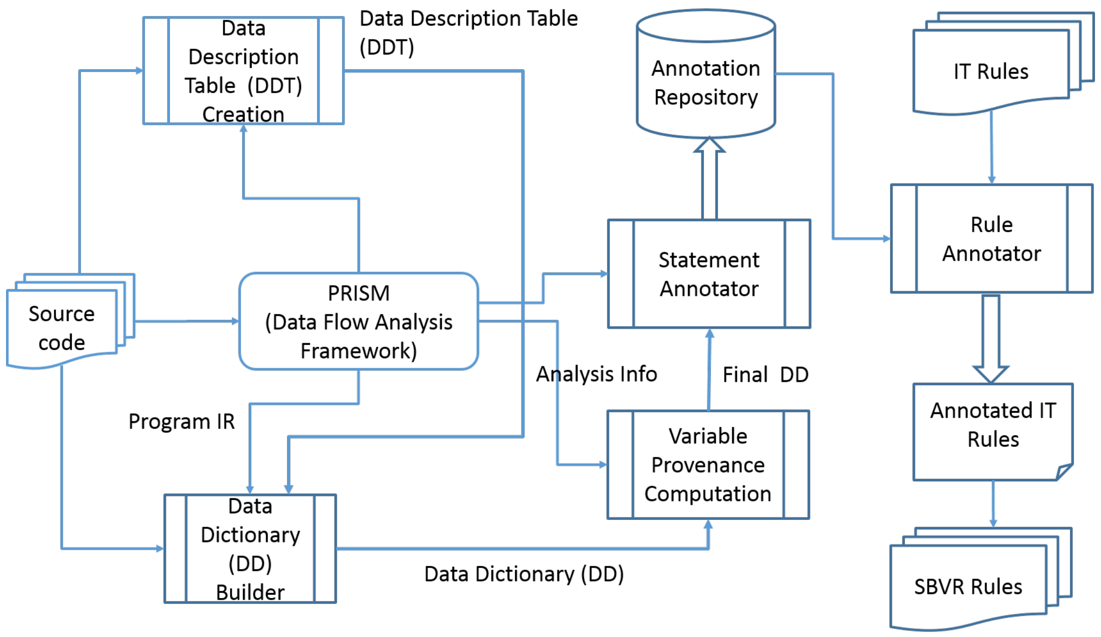

Data Description Table (DDT) Creation
Data Description Table (DDT)
Annotation Repository
IT Rules
Rule Annotator
Source code
PRISM (Data Flow Analysis Framework)
Statement Annotator
Analysis Info
Final DD
Annotated IT Rules
Program IR
Variable Provenance Computation
Data Dictionary (DD) Builder
Data Dictionary (DD)
SBVR Rules

**Figure 2: Block diagram of our approach.**

**Algorithm 3.1:** InitializeDD(*DDT*)

```
 /*Initializes the Data Dictionary using Description Table DDT*/
(1) $DD \leftarrow \phi$
(2) $\forall dt \in DDT$
 /*Get each description table entry */
(3) $< f_i, v_i, d_i > \leftarrow dt$
 /*Initialize each DD equivalence class as -1.*/
(4) $dd =< v_i, d_i, -1 >$
(5) $DD \leftarrow DD \cup dd$
```

AMOUNT. In Step (10), the description of the variable SRC-BALANCE is copied into NEW-BALANCE using *Case 2* of the Algo. 3.3. The variables NEW-BALANCE, SRC-BALANCE, AMOUNT will be put into the same equivalence class.

We have also defined other set of heuristics based on the assignment statements, computational statements (such as $var1 = var2 +$ CONSTANT, $var1 = var2 + / - var3$ and many more). At the end of the algorithm in Step (15), we compute the DD which has variables along with their descriptions grouped in the meaningful equivalence classes.

The updation of DD is guided by the operational rules of logical and computational operators of the computation/assignment statements. The complete algorithm for updating the DD is given in Algo. 3.3.

Let *lhs* be left hand side, *rhs* be right hand side of assignment statement *s*. Let $r$ & $l$ be the variables in *rhs* & *lhs*. The Algo. 3.3 depicts all scenarios that could happen in the creation and updation of equivalence classes.

For complex assignments of the form $x = y \oplus z$, where $\oplus$ can be any operator involving (+) or (−) or (∗). We have applied simple heuristic of copying the equivalence class of *lhs* to *rhs*. For an example, `Simple Interest = (Principle * Rate of Return * Duration) / 100`. Using our heuristics given, the `Principle` and `Interest` both refer to the "*amount*", they can be put together in one equivalence class referred as `Amount`.

## 3.4 Sentence Annotation

The conceptual description of the variables in the program can be used to enable the end users to better understand how the statements are doing what they are expected to do. Our approach makes use of the description generated for the variables in the program using *variable provenance* technique and generates the statement level descriptions for the source code lines based on the custom template provided by the user.

Statement Dictionary (SD) is defined similar to the DD to store the description of all the statements of the given application. The

```
/*Computes the variable provenance for given program P*/
Algorithm 3.2: ComputeVariableProvenance($P$)

 /*Creates a description Table consisting File/DB/Constant*/
(1) $DDT \leftarrow$ CreateDescriptionTab($P$)
 /*Create and Initialize DD using Algorithm III.1*/
(2) $DD \leftarrow$ InitializeDD($DDT$)
 /*Eliminates duplicates*/
(3) $DD \leftarrow$ NormalizeDD($DD$)
 /*Get input/output domain variables from the program P*/
(4) $DV \leftarrow$ GetDomainVar($P$)
(5) $ODV \leftarrow$ GetOutputDV($DV$)
(6) $IDV \leftarrow$ GetInputDV($DV$)
 /*For each o/p domain variable compute backward data slice*/
(7) $\forall o \in ODV$
(8) $BSlice \leftarrow$ ComputeBackwardDataSlice($o$)
(9) $\forall stmt \in BSlice$
 /*Update Data dictionary for that statement*/
(10) UpdateDD($stmt, DD$)
 /*For each i/p domain variable compute forward data slice*/
(11) $\forall i \in IDV$
(12) $FSlice \leftarrow$ ComputeForwardDataSlice($i$)
(13) $\forall stmt \in FSlice$
 /*Update Data dictionary for that statement*/
(14) UpdateDD(stmt,DD)
(15) $return(DD)$
```

SD is formally defined as: SD = $\{< s, d >\}$, where $s$ is the statement in the program, $d$ is the computed statement level description (annotation).

```
<method name="generateMove">
    <arguments>
        <argument type="String" name="@1"/>
        <argument type="String" name="@2"/>
    </arguments>
    <rtype type="String"/>
    <code>
        <order>
            <item type="@1"/>
            <item type="IS␣MOVED␣INTO"/>
            <item type="@2"/>
        </order>
    </code>
</method>
```

**Figure 3: Sample template file for Program Annotation.**

A sample template file is shown in Figure 3 to provide as a guidance to generate description for MOVE statements in COBOL. This sample program takes two arguments represented by "@1" and "@2". The tag sequence <order> </order> is used to generate the description of the statement in selected order. Consider the example shown in Fig. 4, the descriptions for all MOVE statements in program are generated using the above mentioned part of template file and data dictionary that has been built.

Consider the line 012600 in Fig. 4, we get annotations for the variables L-ACCOUNT-ID and SRC-AC-ID as "*Account identifier-Primary key*" and "*Source account id*" from the data dictionary. The variables L-ACCOUNT-ID and SRC-AC-ID gets mapped to arguments "@1" and "@2" respectively. The template defined order is used to generate the method "*generateMove*". The description gets created as "*Account identifier-Primary key* IS MOVED INTO *Source account id*". Just as we have method "***generateMove()***" in our template to generate descriptions for MOVE statements, we have methods to generate descriptions for other code constructs.

The statement level annotations are generated using Algo. 3.4 after computation of all annotations of the known variables. Typically the program contains statements such as assignment (e.g MOVE statements in **COBOL**), conditional expressions (If...Then...), loop statements (For, While), *i/o* statements (READ, WRITE). We can use the descriptions of variables from our data dictionary and the knowledge about type of the statement to annotate the source lines of the program. Our approach provides the way to generate the custom defined descriptions for various code constructs. It takes a template file and data dictionary; and computes statement descriptions. Template file enlists all the possible types (assignment statements, conditional expressions, loop expressions, etc.) of statements in the program and what type of description to generate for the particular statement. As it uses the template file for the description computation, these statement annotations can be generated on the fly (dynamically).

**Algorithm 3.3:** UPDATEDD($s$, $DD$)

```
/*Updates the Data Dictionary for given assignment statement s*/
$r \leftarrow RHS(s)$
$l \leftarrow LHS(s)$
$rhs \leftarrow < r, d_r, eq_r >$
$lhs \leftarrow < l, d_l, eq_l >$
case 1
    if $rhs \in DD$ & $lhs \notin DD$ & $eq_r \neq -1$
    /*put the lhs description and equivalence class same as rhs*/
    then
        $lhs \leftarrow < l, d_r, eq_r >$
        $DD \leftarrow DD \cup lhs$
case 2
    if $rhs \in DD$ & $lhs \notin DD$ & $eq_r = -1$
    /*put the lhs description and equivalence class same as rhs.
    We generate new Equivalence class and assign it to both rhs and lhs*/
    then
        $eq_{new} \leftarrow NewEQ()$
        $rhs \leftarrow < r, d_r, eq_{new} >$
        $lhs \leftarrow < l, d_r, eq_{new} >$
        $DD \leftarrow DD \cup lhs$
case 3
    if $lhs \in DD$ & $rhs \notin DD$ & $eq_l \neq -1$
    /*put the rhs description and equivalence class same as lhs*/
    then
        $rhs \leftarrow < r, d_l, eq_l >$
        $DD \leftarrow DD \cup rhs$
case 4
    if $lhs \in DD$ & $rhs \notin DD$ & $eq_l = -1$
    /*put the rhs description and equivalence class same as lhs.
    We generate new Equivalence class and assign it to both lhs and rhs*/
    then
        $eq_{new} \leftarrow NewEQ()$
        $rhs \leftarrow < r, d_l, eq_{new} >$
        $lhs \leftarrow < l, d_l, eq_{new} >$
        $DD \leftarrow DD \cup rhs$
case 5
    if $lhs \notin DD$ & $rhs \notin DD$
    /*put both lhs rhs with null ($\phi$) description.
    We generate new Equivalence class and assign it to both lhs and rhs*/
    then
        $eq_{new} \leftarrow NewEQ()$
        $d_r, d_l \leftarrow \phi$
        $rhs \leftarrow < r, d_r, eq_{new} >$
        $lhs \leftarrow < l, d_r, eq_{new} >$
        $DD \leftarrow DD \cup rhs$
        $DD \leftarrow DD \cup lhs$
case 6
    if $lhs \in DD$ & $rhs \in DD$ & $eq_r \neq -1$ & $eq_l = -1$
    /*Change the equivalence class of lhs from -1 to equivalence class of rhs*/
    then
        $lhs \leftarrow < l, d_l, eq_r >$
case 7
    if $rhs \in DD$ & $lhs \in DD$ & $eq_l \neq -1$ & $eq_r = -1$
    /*Change the equivalence class of rhs from -1 to equivalence class of lhs*/
    then
        $rhs \leftarrow < r, d_r, eq_l >$
case 8
    if $rhs \in DD$ & $lhs \in DD$ & $eq_l = -1$ & $eq_r = -1$
    /*Genearte new Equivalence class and assign it to the lhs and rhs*/
    then
        $eq_{new} \leftarrow NewEQ()$
        $rhs \leftarrow < r, d_r, eq_{new} >$
        $lhs \leftarrow < l, d_l, eq_{new} >$
case 9
    if $rhs \in DD$ & $lhs \in DD$ & $eq_l \neq -1$ & $eq_r \neq -1$ & $eq_l \neq eq_r$
    /*Genearte new Equivalence class and assign it to the lhs and rhs.
     Update Equivalence class of all members of lhs and rhs*/
    then
        $eq_{new} \leftarrow NewEQ()$
        $R \leftarrow GetMembers(eq_r)$
        $\forall r \in R, rhs \leftarrow < r, d_r, eq_{new} >$
        $L \leftarrow GetMembers(eq_l)$
        $\forall l \in L, lhs \leftarrow < l, d_l, eq_{new} >$
```

## 3.5 IT Rule Annotation

The IT Rule Extraction [17], [20], [9], [21], [12] from large legacy set source code consists of source code lines as a *rule*. The IT rule annotation is explained in the Algo. 3.5.

```
012500      IF  ACCT-RETURN-CODE  =  EXISTING-ACCOUNT                       01250009
012600           MOVE  L-ACCOUNT-ID  OF  ACCOUNT-COPY  TO  SRC-AC-ID         01260009
012700           MOVE  L-ACCOUNT-BALANCE  OF  ACCOUNT-COPY  TO  SRC-BALANCE  01270009
```

**Figure 4: A sample COBOL program.**

```
/*Annotates the Program (P) using Data Dictionary DD and return SD*/

Algorithm 3.4: ANNOTATEPROGRAM($DD, P$)

 /*Get all statements in the Program P*/
 (0)$SD = \phi$
 (1)$stmtList \leftarrow GetAllStmt(P)$
 /*Get each statement from the statement list*/
 (2) $\forall st \in stmtList$
 /*Get list of variables from the statement*/
 (3) $varList \leftarrow getVariable(st)$
 /*get description of each var from DD.*/
 (4) $\forall var \in varList$
 (5) $d_{var} = getDescription(var)$
 /*Load template of annotation generator*/
 (6) $loadTemple()$
 /*generate statement level annotation*/
 (7) $d_{stmt} \leftarrow generateDescription(stmt)$
 /*update the statement dictionary*/
 (8) $SD \leftarrow SD \cup < stmt, d_{stmt} >$
```

```
/*Annotates the IT Rules (R) using Statement Dictionary SD and return annotated rules (AR)*/

Algorithm 3.5: ANNOTATEITRULES($R$)

 (0)$AR = \phi$
 (1)$itRuleList \leftarrow R$
 /*Get each IT rule from the IT rule list*/
 (2) $\forall r \in itRuleList$
 /*Get each statement from the rule r*/
 (3) $AD = \phi$
 (4) $\forall s \in r$
 /*get description of each statement s from SD.*/
 (5) $d_s = getDescription(SD, s)$
 /*create annotated description*/
 (6) $AD \leftarrow AD \cup d_s$
 /*update the annotated rules*/
 (7) $AR \leftarrow AR \cup < r, AD >$
 /*return the annotated rules set*/
 (8) $return(AR)$
```

In this phase, each IT rule from the extracted IT rule set is annotated automatically using SD as shown in Steps (4-6) of the above algorithm. In the end, the annotated rules are populated Step (7) and captured in a rule store for further access.

## 3.6 SBVR Rule Creation

The SBVR is a OMG standard. The rules are created by applying the mappings defined by our earlier research [7]. We assume that the IT Rules have been identified by the domain experts from the legacy systems. These IT Rules are annotated using our earlier steps. In this step, we also use domain expert to review and authorize the rules to be processed further. These rules are subsequently translated using the heuristic rules that are defined in [7]. These heuristic rules are defined on the structure of the rule sentence parsed using Stanford dependency parser [13].

## 4 PRELIMINARY EVALUATION

We have implemented the tool `ProgAnnotator` to depict our general approach of IT rule annotation using *variable provenance*. The tool is implemented as an Eclipse Plug-in. In this section, we describe preliminary evaluations conducted using `ProgAnnotator`. We considered 3 different subjects. The details of the subjects are shown in Tab. 3.

The Subject 1 is a UK Taxation policy case study program written in COBOL. It has 500 lines of code, 2 database tables, and 28 input-output (I-O) variables in the program. The Subject 2 is a Fund Transfer case study for a bank written in COBOL. It has 1,200 lines of code, 9 database tables, 9 source files, and 62 I-O variables in the program. The Subject 3 is a segment of Large legacy Insurance having 80K LOC, and 2021 I-O variables.

We considered two qualitative aspect for the experiments, (1) effectiveness of the approach in creating meaningful statement level descriptions using *variable provenance*, and (2) we have used extracted IT Rules and measured the accuracy of the creation of annotated IT rules.

The tool created statement annotation in `SD` with 97.4% for subject-1. For subject-2, the accuracy was 95% in building statement level descriptions. We have manually verified the correctness of the generation. The subject-3 is a segment of the real-life insurance system code. The statement annotation could not be verified manually due to sheer size of the source code.

`ProgAnnotator` tool annotates 487 lines correctly out of 500 lines of code for subject-1 and for subject-2, 1165 lines out of 1200 lines of code are annotated correctly. This subject has 12 IT rules and 11 IT rules have been annotated accurately (with 91.67% accuracy). Both subjects had variety of COBOL language statements such as assignment statements (`MOVE`, `SET`), conditional expressions (`If..Then`, `Switch case`), loop constructs (`while`), input/output statements (`READ, WRITE`), function calls (`CALL`) etc. The annotated IT rule statements for subject-2 and subject-3 is 94.12% and 94.17% respectively. The results are shown in Tab. 4. All these annotated IT-Rules are successfully translated to `SBVR` rules.

## 5 RELATED WORK

Program comprehension and maintenance of legacy systems is a never ending industrial problem. Gail Murphy et al. proposed Software Reflexion models [15] to view the source code from the lenses of high level models defined by an engineer to understand the systems. This work uses an engineer defined high level model and a model extracted from source code (such as a call graph or an inheritance hierarchy) and defines a declarative mapping between the two models. A software reflexion model is then computed to see where the engineer's high level model alignment with the source model. Their studies on NetBSD and Microsoft Excel confirms the applicability/reliability of the approach. As the source models have been created manually by a domain expert, the applicability of such techniques to annotate IT rules from business system becomes a non trivial activity. This approach is a good way to explain the structural aspects of the system at high level, but not as an application towards annotating business rules. The work by A. van Deursen and L. Moonen [19] shows hypertext-based program understanding. This approach achieves a new levels of abstraction by using inferred type information for cases where the subject software system is written in a weakly typed language such as COBOL. Their work is very close to our adapted approach. In this approach, the authors inferred a type based on the usage of a variable. They used various inferencing rules on the variables to group them in to a cluster. This approach starts with assigning every variable a unique primitive type and creates equivalences between these types based on the usage such as:

- If variables are compared using some relational operator, it is inferred that they belong to same type
- If an expression is assigned to a variable, the type of the variable must be that of the expression.

The detailed inference rules used are based on the analysis of the use of variables [18]. The inferencing rules used poses a serious threat to clustering of variables. For an example, a naively written program in which a same variable is used in multiple places in different contexts to hold different types of values may lead to the incorrect grouping of variables. The other limitation of this approach was handling of the minimal set of literals included in certain types. In our approach, we cluster the variables based on control flow and data flow analysis making grouping of variables more precise than the approach shown by above authors.

Provenance based techniques have been adopted by researchers to understand the legacy systems over the years. A survey paper by Simmhan et al. [16] talks about several research efforts in this direction. The *VisTrails system* [6] shows how the availability of provenance supports reflective reasoning. This is very crucial task for users who perform exploratory tasks. The VisTrails tool provides a comprehensive provenance infrastructure for computational tasks. Traditionally, the workflows are used to automate repetitive tasks, but *VisTrails* uses provenance for exploratory tasks where change is the norm. It uses a change based provenance where the trails of every exploration are maintained. This information then is used for workflow evolution. The generated workflows can be revisited by scientists to reproduce, validate, and collaborate and perform knowledge sharing. At large, the *VisTrails* focuses on exploratory tasks such as simulations, visualization and data mining. On a similar line of work on the scientific workflow management just as *VisTrails*, a scientific workflow system (Kepler) [4], aims to keep track of all aspects of provenance in scientific workflows: in workflow evolution, data and process provenance, and efficient management and usage of collected data. Their approach differs from *VisTrails* in a way that they provide a design of a provenance collection framework that is highly configurable. This provenance collection facility is parametric and customizable which a user may use to limit the granularity of the collected data.

The dynamic analysis techniques coupled with instrumentation and log generation [3], [5] have the difficulty of generating the right amount and quality of log information, and rely on adequacy of test-data, making it very difficult to ensure generation of all data provenances. The class summaries [8] can help only in understanding the content of Java programs but not the relation

**Table 3: Details of the subjects used for experimentation**

| No | Subject Name | LOC | No. of Tables | No. of Domain Vars |
|---|---|---|---|---|
| 1 | UK Taxation | 500 | 2 | 28 |
| 2 | Fund Transfer | 1200 | 9 | 62 |
| 3 | Large Insurance (segment) | 80500 | 5 | 2021 |

**Table 4: Results of the study conducted.**

| No | Subject Name | Statement Annotated Accuracy | No. of IT Rules Considered | IT Rule Annotation Accuracy |
|---|---|---|---|---|
| 1 | UK Taxation | 97.40% | 12 | 91.67% |
| 2 | Fund Transfer | 95% | 17 | 94.12% |
| 3 | Large Insurance (segment) | NA | 103 | 94.17% |

among the classes and methods. Therefore, we used the concept of variable provenance [11] in the context of a software process or procedure [14]. There have been many attempts made in making legacy programs comprehensible. One of the ways to get the better understanding of the legacy systems is using the provenance. While there is a voluminous amount of work done both on an attempt to understand legacy systems better as well on the data provenance, our approach differs from the existing work in a way that we used the concept of provenance descriptions of the variables [11] (actually the provenance of a various types of data described as natural text held by variables at different times) to annotate the source code lines which constitutes the rules implemented within a system. We use provenance information on variables and program analysis techniques to create clusters of related variables (variables that are used in similar patterns or the situations throughout the program). This in turn helps us in annotating the source code with meaningful natural language like descriptions. Using data-flow analysis and control-flow analysis, our approach computes programme slices for the IT rules extraction. Our approach to use provenance of variables to give meaningful descriptions and annotate the extracted IT rules is a novel technique in a gamut of already existing works on understanding legacy systems better. Our adapted approach creates provenance information statically, rather than dynamically, sets it apart from nearly all previous work. Most previous work, for example scientific data provenance, using tools such as *Kepler* [4] and *VisTrails* [6], creates a detailed data provenance on the fly as the program executes. We on the other hand compute provenance statically and automatically based upon flow analysis such as reaching definitions, slicing, etc. Our approach is novel and doesn't require a program execution (dynamic analysis) for collection and use of provenance.

## 6 CONCLUSIONS

We have explored the idea of *variable provenance*, which creates text descriptions for all 'relevant' variables in the source code. For the sample set of programs, we have observed that the annotations generated for the variables, conditional expressions and the statements are by-and-large meaningful for the developers to understand the program logic. We used these annotated statement descriptions to annotate the embedded IT rules for the sample and near-real-life programs. We annotated the IT rules with 91% to 94% accuracy.

Following are the challenges that we encountered and which we like to take forward:

- Critical assumption is the availability of descriptions for persistent entities and constants used in the source code. While descriptions for database tables and columns are usually available, it is a challenge to procure/create descriptions for Files and Constants.
- Combining provenance of multiple variables in a complex calculation, especially under multi-conditions is a non-trivial activity.
- Multiple provenances may be assigned to some variables. This is because developers may use the same variable in multiple contexts, usually in non-overlapping life spans. To overcome the problem, annotation of variable can be stored with the reference rather than its declaration. This decision reflects a tradeoff between excessive storage and loss of precision of provenance.
- Annotations attached to variables can blow-up due to imprecise program analysis. A trade-off between imprecision and execution time is the result of the choice of context- and path-insensitive analysis versus context- and path-sensitive analysis.
- For abstractions like loops, functions, we need to study and evolve what kind of descriptions can be synthesized from those of the contained statements.
- To generate description we rely on operators such as arithmetic, assignment, and the type of statement. While these descriptions are good enough to capture the essential at initial level, the abstract level description of code fragments will be more meaningful and to generate such abstract description is not trivial activity.

## REFERENCES

[1] [n. d.]. Semantics Of Business Vocabulary And Rules (SBVR). http://www.omg.org/spec/SBVR/. ([n. d.]). [Online; accessed 19-July-2015].

[2] 2008. *Prism: Static data and control Flow analysis workbench.* Technical Report. Tata Consultancy Services Ltd., Pune, India.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. 2006. Provenance Collection Support in the Kepler Scientific Workflow System. In *Proceedings of the 2006 International Conference on Provenance and Annotation of Data (IPAW'06).* Springer-Verlag, Berlin, Heidelberg, 118–132. https://doi.org/10.1007/11890850_14

[5] Roger S. Barga and Luciano A. Digiampietri. 2006. Automatic Generation of Workflow Provenance. In *Proceedings of the 2006 International Conference on Provenance and Annotation of Data (IPAW'06).* Springer-Verlag, Berlin, Heidelberg, 1–9. https://doi.org/10.1007/11890850_1

[6] Louis Bavoil, Steven P. Callahan, Patricia J. Crossno, Juliana Freire, and Huy T. Vo. 2005. VisTrails: Enabling interactive multiple-view visualizations. In *In IEEE Visualization 2005.* 135–142.

[7] Abhidip Bhattacharyya, Pavan Kumar Chittimalli, and Ravindra Naik. 2017. An Approach to Mine Business Rule Intents from Domain-specific Documents. In *Proceedings of the 10th Innovations in Software Engineering Conference (ISEC '17).* ACM, New York, NY, USA, 96–106. https://doi.org/10.1145/3021460.3021470

[8] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends databases* 1, 4 (April 2009), 379–474. https://doi.org/10.1561/1900000006

[9] Chia-Chu Chiang. 2006. Extracting business rules from legacy systems into reusable components. In *System of Systems Engineering, 2006 IEEE/SMC International Conference on.* 6 pp.–. https://doi.org/10.1109/SYSOSE.2006.1652320

[10] Pavan Kumar Chittimalli and Kritika Anand. 2016. Domain-independent Method of Detecting Inconsistencies in SBVR-based Business Rules. In *Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems (ForMABS 2016).* ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/2975941.2975943

[11] Pavan Kumar Chittimalli and Ravindra Naik. 2014. Variable Provenance in Software Systems. In *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering (RSSE 2014).* ACM, New York, NY, USA, 9–13. https://doi.org/10.1145/2593822.2593826

[12] A.B. Earls, S.M. Embury, and N.H. Turner. 2002. A method for the manual extraction of business rules from legacy source code. *BT Technology Journal* 20, 4 (2002), 127–145. https://doi.org/10.1023/A:1021311932020

[13] Vijay Krishnan and Christopher D. Manning. 2006. An Effective Two-stage Model for Exploiting Non-local Dependencies in Named Entity Recognition. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics (ACL-44).* Association for Computational Linguistics, Stroudsburg, PA, USA, 1121–1128. https://doi.org/10.3115/1220175.1220316

[14] Simon Miles. 2010. Automatically Adapting Source Code to Document Provenance. In *Provenance and Annotation of Data and Processes*, DeborahL. McGuinness, JamesR. Michaelis, and Luc Moreau (Eds.). Lecture Notes in Computer Science, Vol. 6378. Springer Berlin Heidelberg, 102–110. https://doi.org/10.1007/978-3-642-17819-1_13

[15] Gail C. Murphy, David Notkin, and Kevin Sullivan. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-level Models. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '95).* ACM, New York, NY, USA, 18–28. https://doi.org/10.1145/222124.222136

[16] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2005. A Survey of Data Provenance in e-Science. *SIGMOD Rec.* 34, 3 (Sept. 2005), 31–36. https://doi.org/10.1145/1084805.1084812

[17] H.M. Sneed. 2001. Extracting business logic from existing COBOL programs as a basis for redevelopment. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on.* 167–175. https://doi.org/10.1109/WPC.2001.921728

[18] A. van Deursen and L. Moonen. 1998. Type inference for COBOL systems. In *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on.* 220–230. https://doi.org/10.1109/WCRE.1998.723192

[19] Arie van Deursen and Leon Moonen. 2006. Documenting software systems using types. *Science of Computer Programming* 60, 2 (2006), 205 – 220. https://doi.org/10.1016/j.scico.2005.10.006 Special Issue on Software Analysis, Evolution and, Re-engineering.

[20] Chengliang Wang, Yaxin Zhou, and Juanjuan Chen. 2008. Extracting Prime Business Rules from Large Legacy System. In *Computer Science and Software Engineering, 2008 International Conference on*, Vol. 2. 19–23. https://doi.org/10.1109/CSSE.2008.497

[21] Xinyu Wang, Jianling Sun, Xiaohu Yang, Zhijun He, and S. Maddineni. 2004. Business rules extraction from large legacy systems. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on.* 249–258. https://doi.org/10.1109/CSMR.2004.1281426