Journal of International Information Management

Volume 12 | Issue 1 Article 4

2003

The development of a business rules engine: A condition-action rule algorithm for finite static lists

Terri D. Giddens
Texas Tech University

Kevin E. Gaasch
Panhandle Plains Student Loan Center

The development of a business rules engine: A condition-action rule algorithm for finite static lists

Terri D. Giddens
Texas Tech University

Kevin E. Gaasch
Panhandle Plains Student Loan Center

ABSTRACT

An on-going theme in Information Systems research is the methods by which business rules are gathered and implemented. Additionally, many efforts have been made to develop reusable algorithms for processing business rules to reduce system development, testing, and maintenance time. The objective of this paper is to present a reusable algorithm for condition-action rules that are applied to finite static lists. More importantly, the algorithm is generalized for complex rules that are complicated by differences in user authorizations and other dependencies.

INTRODUCTION

The process of gathering system requirements in a business domain involves the definition of business rules, or more particularly, production rules or condition-action rules. These rules may be thought of as intrinsic bits of knowledge concerning a particular system's domain. A condition-action business rule is any statement that can be put in the form of "if - then". A rule is defined as: "A statement that defines or constrains some aspect of the business. It is intended to assert business structure, or to control or influence the behavior of the business" (Perkins, 2000). One major and continuing problem in software development is the implementation of business rules as hard-coded elements that are often replicated throughout many applications. As a consequence, they are neither maintainable nor reusable (Belderrain, 2002) (Rouvellou et al., 2000). Business rules often change during an application's life cycle, and subsequent changes of these business rules may have an adverse impact on the application. Analysts or programmers may change or add rules without a full understanding of the existing rules or their replication (Grosoff et al., 2000).

The use of generic, encapsulated, reusable business rules algorithms has been recognized as a fundamental need in corporate software development. Over the past thirty years, many papers have presented architectural frameworks that attempt to make the implementation of business rules more flexible and designed for change (Grosoff et al., 2000)(Shao et al., 2001). Many of these include case-based reasoning, externalized or componentized rules, neural networks, and knowledge-based inference engines. The goal for each is the development of a "rules engine" that processes business rules and aids in the development, testing, and maintenance of complex systems. The need for a rules engine is emphasized by the fact that most business rules have low stability, high complexity, and require a high effort to enforce (Rosca and D'Atillio, 2001)(Rosca et al., 1995). The high complexity is explained by the interdependencies that exist on other business rules. It is further argued that rule encapsulation and externalization significantly enhances maintainability (Belderrain, 2002) (Rouvellou et al., 2000).

This paper presents a reusable algorithm to be used in a business rules engine. The algorithm is for one specific type of a business rule; it is to be used on finite static lists that have a multiple set of rules triggered by additional dependencies on other fields and differences in authorization that have been defined for various classes of users. This paper begins with an overview of the literature on Condition-Action Rules. This is followed by an industry example to be used to facilitate the presentation of the algorithm. The first iteration of the algorithm in its simplest form, a finite static list assigned business rules for a single user authorization, is then presented. In the second iteration, the algorithm is modified for more complex business rules that are affected by multiple user authorizations. Because business logic is often further complicated by multiple dependencies on other data input fields, the third iteration of the algorithm is finally generalized for n-dimensional dependencies. The benefit of a system that uses this algorithm is then discussed.

CONDITION-ACTION RULES

Condition-Action Rules are based on action assertions. Action assertions specify constraints on the data that is produced or modified by the actions. A "condition" is a test that is used to determine whether to perform certain actions or test other action assertions. One type of action assertion is an "integrity constraint". An integrity constraint is an assertion that tests for a valid state that must always be true. Integrity constraints are often complicated by the variations in the business rules affected by user "authorization". Authorization specifies user permissions to perform certain actions; the integrity constraint may have a different set of rules depending on the authorization assigned to a particular user.

A significant portion of business logic is associated with validation of data. Most data validation is done through integrity constraints. All data entry fields have a value that is validated prior to saving the data. The validation state must be true in order for the data to be saved. The following are various classes of action assertions that are applied to simple data validation:

- ***Required fields***. On a data entry screen some fields are required to be entered and will not save unless there is something in the field.

- *Field size limits.* All fields that are being saved to a database must not exceed the field length specified in the structure of the database.
- *Valid Dates*: All dates must be a valid date. Some systems require additional date field validations.
- *Upper/Lower Limits.* Some fields may not be valid if they are out of bounds on a lower or upper limit.
- *Dependencies.* Some fields need additional validation depending on the value of another field.
- *Finite Static Lists.* Some fields have a finite set of values that the field can contain. This is typically displayed in a pull-down (combo) list or list box where there are a finite number of choices that can be made.

Business logic defined for data input usually requires a combination of the above assertion types. For example, a required field may have a length constraint and must be one of a certain set of values from a finite static list. The nested "if" statements in a program with multidimensional business logic can get very complicated and hard to read. Condition-Action rules for data validation need to be flexible and easy to modify, program, and test.

INDUSTRY EXAMPLE

To facilitate the discussion, an industry instance where the algorithm has successfully been implemented is presented. The following is a brief overview of the industry example.

In the student loan industry, student loan applications are received and processed. The "origination" of a loan involves coordination between the school, lender, borrower (student or parent) and guarantor.

The process that a borrower goes through to get a student loan is:

1. A borrower fills out an application and selects a school and lender.
2. The application is submitted to the school, lender, and guarantor.
3. Before a loan is originated from the application, the application must be guaranteed by a federal guarantee agency. Once a notice of guarantee is received from the guarantee agency, the lender's loan department "matches" it to an application and it becomes a "loan".
4. Included on the loan are disbursements (dates and amounts) that are set by the school. The lender processes the disbursements according to these dates and amounts.
5. After the loan is fully disbursed, the loan is sent to servicing where interest accrues and payments are solicited.

Applications, loans, and disbursements can have one of a fixed number of statuses. The business logic for these finite static lists defines how each of the statuses can be changed. A status can be changed to another value based on its initial value, the type of authorization the user has been assigned, the type of loan that it is, and the amount requested.

For this example, the status of an application will be used. An application can have the following finite number of statuses:

- Matched – It has become a "loan" because a guarantee has been received for it.
- Pending – The application is waiting for a guarantee to be "matched" to it.
- Hold – Something is wrong with the application – it cannot be "matched" in this status.
- Canceled – The application has been canceled by the Borrower, School or Guarantor.

The inclusion of multiple user authorizations makes the business logic multi-dimensional. The valid changes from one value to another are further complicated by the authorization granted to the user. A user is able to change the status of the application according to the authorization he/she has been assigned. A "supervisor" user is able to change from any status to any other status. A "guest" is not able to change the status. A "staff" user can change the status of the application according the following rules:

- Once matched, the application cannot change status.
- A pending application can be placed on hold.
- A pending application can be matched.
- An application on hold can be removed from hold and put back to pending
- A pending application can be canceled.
- An application on hold can be canceled.
- Once canceled, the application cannot change status.

To further complicate this business logic, this finite static list has dependencies on other fields. For example, depending on the type of application or the requested amount, the application must be on hold if some data entry fields are blank.

Business logic can become very complex and therefore extremely hard to program, test, and maintain. Pseudo code for the algorithm for the above business rules would be:

```
Function Validate_Status   (initial_status, changed_status, UserAuthorization)
  If UserAuthorization  =  "staff" then
      If initial_status = "Hold" then
         If changed_status = "Pending" then  valid = true;
         If changed_status = "Matched" then valid = false;
         If changed_status = "Canceled" then valid = true;
      End if //status = Hold"
      If initial_status = "Pending" then
         If changed_status = "Matched" then valid = true;
         If changed_status = "Canceled" then valid = true;
```

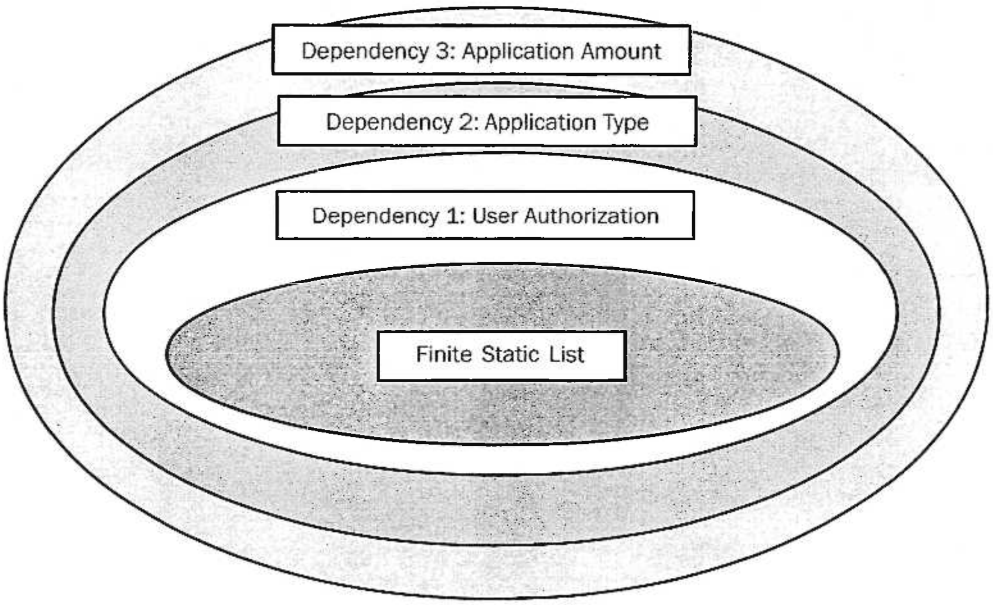
```
        If changed_status = "Hold" then valid = true;
      End if //status = Pending"
      If initial_status = "Matched" then
        If changed_status = "Pending" then valid = false;
        If changed_status = "Canceled" then valid = false;
        If changed_status = "Hold" then valid = false;
      End if //status = Matched"
      If initial_status = "Canceled" then
        If changed_status = "Pending" then valid = false;
        If changed_status = "Matched" then valid = false;
        If changed_status = "Hold" then valid = false;
      End if //status = Canceled"
    End if //UserAuthorization = staff
    // ******** repeat above for other UserAuthorizations *****************
    Return valid
  Function End
```

In the above scenario, the validation of this finite static list of 4 items for one user authorization requires 2^4 or 16 "if" statements. These 16 statements would need to be duplicated for each user authorization. For this scenario, the logic to support the dependency on the type of loan and the amount of the loan would also have to be added. Beyond becoming a programming nightmare, this is very difficult to test and maintain as the business rules change.

CONCEPTUALIZATION OF BUSINESS LOGIC COMPLEXITY OF A FINITE STATIC LIST WITH DEPENDENCIES

The following conceptual diagram shows the finite static list and its dependency on other values that complicate the business logic. In the center is the finite static list. The first dimension around it shows its dependency on user authorization. Valid status changes are defined by the type of user authorization permitted. The business logic is further depicted by showing the finite static list's dependencies on the application type and the application amount in the remaining dimensions. Depending on the application type and the application amount, the business rules that dictate valid status changes differ.

Figure 1. Conceptual Diagram of Finite Static List with Dependencies


Dependency 3: Application Amount
Dependency 2: Application Type
Dependency 1: User Authorization
Finite Static List

To further complicate matters, complex business logic is often viewed by the programmer as a challenge. Given a set of business rules, a clever programmer will attempt to simplify the logic into a minimal set of intertwining nested if statements using some sort of boolean logic reduction techniques. This over-simplification usually makes the program hard to maintain. If the logic is not clearly documented or easy to read, subsequent changes to the code may be made incorrectly and have adverse effects on logic of the application.

FINITE STATIC LIST ALGORITHM

To simplify the explanation of the algorithm, discussion of the algorithm will first be given using only the business rules for a user of the system who has been given "staff" system authorization. This will be referenced as a finite static list having business rules defined with a one-dimensional dependency. After this initial presentation, the algorithm will be generalized to handle multi-dimensional complex condition-action rules that incorporate multiple dependencies as illustrated in the business rules for the industry example.

The following table defines the algorithm in a generic form in the first column. For clarification, the algorithm is applied to the industry example and is given in the second column.

Table 1. Simple algorithm with single user authorization

Generic Form	Applied Form
n: number of elements in Finite Static List	n = 4 Statuses: matched, pending, hold, canceled

I = A single Item in a Finite Static List,
$I_i \in [I_0, I_1, \ldots I_{n-1}]$, $0 \le i < n$, and n is finite
 	
F = Finite Static List of n Items
	I_0 = matched, I_1 = pending,
	I_2 = hold, I_3 = canceled
	
	F = [matched, pending, hold, canceled]

I_i = Initial value selected from F, $0 \le i < n$ I_j = Changed value selected from F, $0 \le j < n$, $I_i <> I_j$ $E_{ij} \in [(I_0, I_1), (I_0, I_2), \ldots (I_{n-2}, I_{n-1})]$ = A combination of 2 Items (I_i, I_j) where each E_{ij} represents a change of value from one Item in the Finite Static List (I_i) to another element in the Finite Static List (I_j), $i <> j$	Status initial value is either matched, pending, hold, or canceled. The status is changed to another value. A change to the same value is not relevant. E_{12} = [pending, hold] implies that the initial value was "pending" and changed to "hold". A change from "hold" to "hold" is not relevant.
$b = 2^n$ = number of bits needed to represent all possible (E_{ij}) state changes.	$b = 2^4 = 16$. There are 16 possible state changes from one item in the list to another.
S = A set of b bits, where each bit in S maps to each E_{ij}	S contains 16 bits, each representing one state change. For the above business rules, S = "0000 1011 0101 0000" Bit position 0 = 0 (matched to matched) Bit position 1 = 0 (matched to pending) Bit position 2 = 0 (matched to hold) Bit position 3 = 0 (matched to canceled) Bit position 4 = 1 (pending to matched) Bit position 5 = 0 (pending to pending) Bit position 6 = 1 (pending to hold) Bit position 7 = 1 (pending to canceled) Bit position 8 = 0 (hold to matched) Bit position 9 = 1 (hold to pending) Bit position 10 = 0 (hold to hold) Bit position 11 = 1 (hold to canceled) Bit position 12 = 0 (canceled to matched) Bit position 13 = 0 (canceled to pending) Bit position 14 = 0 (canceled to hold) Bit position 15 = 0 (canceled to canceled)
P_{ij} = Calculated position of bit in S mapped for each E_{ij}. $P_{ij} = n{*}i + j$, $P_{ij} < b$	If changing from pending to hold: Pending: i = 1; hold: j = 2, n = 4 Bit position = 4*1 + 2 = 6

$S(P_{ij})$= Value of bit in S at position P	
$S(P_{ij}) = 0$ if E_{ij} is not a valid state change	If the value at the calculated bit position is a 0, it is not a valid state change.
$S(P_{ij}) = 1$ if E_{ij} is a valid state change	If the value at the calculated bit position is a 1, it is a valid state change.
Condition-Action Rule: Given I_i, determine whether a change to I_j is valid. If a change from I_i to I_j is not valid, do not allow the change. Evaluation:	Given a change from "pending" to "hold": Calculate the bit position: Initial Status (pending) = 1 Changed Status (hold) = 2 Bit Position = 4 * 1 + 2 = 6

V = true if $S(P_{ij}) = 1$
V = false if $S(P_{ij}) = 0$
	If value at bit position "6" = 1 it is a valid change
	If value at bit position "6" = 0 it is not valid

The key to understanding the algorithm is the bit mapping of each possible status change in S. S contains a bit for each possible status change. The first four bits are for an *initial* status of "matched". The next four are for an *initial* status of "pending". The next four are for an *initial* status of "hold". The last four bits are for an *initial* status of "canceled". Each of the bits within the grouping of four represents the *changed* status. Bit position zero in the group of four represents a status *change* to "matched". Bit position one in the group represents a status *change* to "pending". Bit position two in the group represents a status *change* to "hold". Bit position three in the group represents a status *change* to "canceled". Therefore, bit position six in S represents a change from pending to hold. If the bit at that location is set to 1, the change is valid. If the bit at that location is set to 0, the change is invalid.

The following is an object oriented approach to the algorithm using Java. It should be noted that the algorithm can be implemented many different ways using different languages and even XML to identify the rules. The following Java code segments are given just to demonstrate the ease of programming using this algorithm.

Sample Code: StatusChangeValidator.java

```
import java.util.*;

public class StatusChangeValidator {

    private Hashtable itemTable = new Hashtable();

    private int itemCount;
    BitSet validChanges;
    int numStatusItems;

    StatusChangeValidator(int numItems) {
      itemTable = new Hashtable();
      itemCount = 0;
      numStatusItems = numItems;
      validChanges = new BitSet(Math.pow((double)2, (double)numStatusItems);
     }

    public void addItem(String s) {
      itemTable.put(s, String.valueOf(itemCount));
      itemCount++;
     }

    public void addValidStatusChange(String s1, String s2) {
       validChanges.set(numStatusItems *
  Integer.parseInt(itemTable.get(s1).toString())
                  +Integer.parseInt(itemTable.get(s2).toString()));
     }

    public boolean isValid(String s1, String s2)
      return (validChanges.get(numStatusItems*
  Integer.parseInt(itemTable.get(s1).toString())
                  +Integer.parseInt(itemTable.get(s2).toString())));
     }
  }
```

The code identifies a StatusChangeValidator class that incorporates a Hashtable and a BitSet. The Hashtable is used to store the finite static list and the order of each item in the list. The Hashtable is used to "lookup" the item and return its position in the list in order to calculate the bit position in the BitSet. The BitSet is used to store the bits mapped to each possible change from one item in the finite static list to another. The Hashtable represents F and the BitSet represents S.

The StatusChangeValidator class has four methods: the constructor, addItem(), addValidStatusChange(), and isValid(). The constructor method creates the Hashtable and the Bitset and initializes the value of numStatusItems (n). "itemCount" is initially set to 0 and is used as a counter as items are added to the Hashtable. Initially, all bits in the Bitset are set to 0 to represent invalid status changes.

The method addItem() is used to add an item to the Hashtable. The method addValidStatusChange is used to set a bit in the Bitset to represent a valid status change. The method isValid() is used to calculate the bit location and return the bit (either true or false) from the Bitset.

The following is the client code that uses the StatusChangeValidator:

```
public class TestIt {
 public static void main(String args[]) {
    //create a statuschangevalidator for a finite static list with 4 elements
    StatusChangeValidator scv = new StatusChangeValidator(4);
    // add the 4 Items to the validator
     scv.addItem("matched");
    scv.addItem("pending");
    scv.addItem("hold");
     scv.addItem("canceled");

    //specify the business rules
   scv.addValidStatusChange("pending", "hold");
    scv.addValidStatusChange("pending", "canceled");
    scv.addValidStatusChange("pending", "matched");
   scv.addValidStatusChange("hold", "pending");
    scv.addValidStatusChange("hold", "canceled");


    //validate the rule
    if(scv.isValid(args[0], args[1]))
         System.out.println("allowed");
     else
         System.out.println("not allowed");
  }
}
```

The client code creates an instance of the StatusChangeValidator with four items. Each of the four statuses is added to the StatusChangeValidator. Each valid status change is then defined using addValidStatusChange(). The change is validated by a single "if" statement testing the value of the isValid() method. Running the program requires the user to specify two parameters, the initial status followed by the changed status. To run this sample program, the user would enter "java TestIt pending hold".

The programming of this business logic in a traditional form would have taken sixteen "if" statements that would have been harder to code, test and maintain. Using the "black box" rules-engine algorithm, the code has been simplified to a single "if" statement. The business logic is easy to read and change. To add another valid status change, only one line of code would need to be added. Also, adding another status to the list would also require one only line of code. From a programmer's point of view, the logic is much easier to follow. Consequently, the maintenance programmer would be less likely to jeopardize existing business logic while implementing new rules. From a developer's point of view, a structure is created that tends to enforce its own rules for change while preserving the integrity of the code.

GENERALIZED – MULTIPLE USER AUTHORIZATIONS

The following table defines the algorithm for finite static lists with multiple user authorizations. Once again, for clarification, the algorithm is applied to the industry example and is given in the second column.

Table 3. Generalized algorithm with multiple user authorization

Generic Form	Applied Form
n: number of elements in Finite Static List	$n = 4$ Statuses: matched, pending, hold, canceled

I = A single Item in a Finite Static List,
$I_i \in [I_0, I_1, \ldots I_{n-1}]$, $0 \le i < n$, and n is finite
F = Finite Static List of n Items
	I_0 = matched, I_1 = pending,
	I_2 = hold, I_3 = canceled
	
	F = [matched, pending, hold, canceled]

D = A set of m user authorizations D_k = A single user authorization assigned from m authorizations, $0 \le k < m$	D = [guest, staff, supervisor] $m = 3$ D_0 = guest, D_1 = staff, D_2 = supervisor
I_i = Initial value selected from F, $0 \le i < n$ I_j = Changed value selected from F, $0 \le j < n$, $I_i <> I_j$	Status initial value is either matched, pending, hold, or canceled The status is changed to another value. A change to the same value is not relevant.
$E_{ij} \in [(I_0, I_1), (I_0, I_2), \ldots (I_{n-2}, I_{n-1})]$ = A combination of 2 Items (I_i, I_j) where each E_{ij} represents a change of value from one Item in the Finite Static List (I_i) to another element in the Finite Static List (I_j), $i <> j$ $b = m * 2^n$ = number of bits needed to represent all possible (E_{ij}) state changes for all m authorizations.	E_{12} = [pending, hold] implies that the initial value was "pending" and changed to "hold". A change from "hold" to "hold" is not relevant $b = 3 * 2^4 = 48$. There are 16 possible state changes from one item in the list to another. There are 3 possible user authorizations. Total bits needed is 48.
$E_{ij}(D_k)$ = A state change (E_{ij}) for a given user authorization (D_k).	Each of the 16 state changes must be replicated for each user authorization.
S = A set of b bits, where each bit S_{ijk} maps to each $E_{ij}(D_k)$	S contains 48 bits.
$P_{ij}(D_k)$ = Position of bit in S mapped for each $E_{ij}(D_k)$. $P_{ij}(D_k) = ((n*i + j) + (k * 2^n))$, $P_{ij}(D_k) < b$.	Changing from pending to hold with a user authorization of supervisor: pending: $i = 1$; hold: $j = 2$, $n = 4$ supervisor: $k = 1$ Bit position $= (4*1 + 2) + (1 * 2^4) = 22$

$S(P_{ij}(D_k))$= Value of bit in S at position $P_{ij}(D_k)$	
$S(P_{ij}(D_k)) = 0$ if $E_{ij}(D_k)$ is not a valid state change $S(P_{ij}(D_k)) = 1$ if $E_{ij}(D_k)$ is a valid state change	If the value at the calculated bit position is a 0 it is not a valid state change. If the value at the calculated bit position is a 1, it is a valid state change.
Condition-Action Rule: With a user authorization of D_k, given I_i, determine whether a change to I_j is valid If a change from I_i to I_j is not valid, do not allow the change. Evaluation: V = true if $S(P_{ij}(D_k)) = 1$ V = false if $S(P_{ij}(D_k)) = 0$	Given a user authorization of supervisor and a change from "pending" to "hold": Calculate the bit position: Initial Status (pending) = 1 Changed Status (hold) = 2 Supervisor authorization = 1 Bit Position = $(4*1 + 2) + (1 * 2^4) = 22$ If value at bit position "22" = 1 it is a valid change If value at bit position "22" = 0 it is not valid

The key to understanding the changes made to the algorithm to accommodate the complications added by differing user authorizations is that the initial bit set that represents each possible status change in the finite static list has to be replicated for each user authorization. If there are four items in the list, then there are 2^4 bits required to represent each valid state change. These 16 bits must be replicated for each user authorization. Therefore, if there are three different user authorizations, the number of bits needed would be 16 * 3 or 48. The Java program given would need to be modified slightly to add this second dimension by overloading some of the methods to accommodate additionally passing the user authorization to be used in the calculation of bit positions.

GENERALIZED - MULTIPLE USER AUTHORIZATIONS AND MULTIPLE DEPENDENCIES ON OTHER FIELDS

The following table defines the algorithm for finite static lists with multiple user authorizations and multiple dependencies on other fields. Once again, for clarification, the algorithm is applied to the industry example and is given in the second column. In this example, it is assumed that there are different types of applications and the status change is not only dependent on the authorization granted to the user, but also to the type of application and the amount of the loan request.

Table 3. Generalized algorithm with multiple user authorizations and multiple dependencies

Generic Form	Applied Form
n: number of elements in Finite Static List	n = 4 Statuses: matched, pending, hold, canceled

I = A single Item in a Finite Static List,
$I_i \in [I_0, I_1, \ldots I_{n-1}]$, $0 \leq i < n$, and n is finite
F = Finite Static List of n Items
	I_0 = matched, I_1 = pending,
	I_2 = hold, I_3 = canceled
	
	F = [matched, pending, hold, canceled]

d = number of dependencies D = A set of dependencies on other fields and/or user authorization	d = 3 dependent on user authorization, application type and loan amount D = [user authorization, application type, loan amount]
D_i = A dependency on another field and/or authorization assigned from *d* dependencies, $0 \leq i < d$	D_0 = Dependency on user authorization D_1 = Dependency on application type D_2 = Dependency on loan amount d = 3
$D_i(m)$ = number of distinct possibilities for a dependency	$D_0(m) = 3$ Different user authorizations are guest, staff, and supervisor. $D_1(m) = 2$ Different application types are student initiated or parent initiated. $D_2(m) = 2$ Different loan amounts are those > \$500 and those <= \$500.
D_{ik} = the value (k) chosen from the choices from D_i	D_0 represents the dependencies on the three user authorizations: D_{00}, where k = 0, represents "guest" D_{01} where k = 1, represents "staff" D_{02} where k = 2, represents "supervisor" D_1 represents the dependencies on the two application types: D_{10} where k = 0, represents "student initiated" D_{11} where k = 1, represents "parent initiated" D_2 represents the dependencies on the two loan amounts: D_{20} where k = 0, represents < \$500 D_{21} where k = 1, represents > \$500

I_i = Initial value selected from F, $0 \le i < n$	Status initial value is either matched, pending, hold, or canceled
I_j = Changed value selected from F, $0 \le j < n$, $I_i <> I_j$	The status is changed to another value. A change to the same value is not relevant
$E_{ij} \in [(I_0,I_1), (I_0, I_2), \ldots (I_{n-2}, I_{n-1})]$ = A combination of 2 Items (I_i, I_j) where each E_{ij} represents a change of value from one Item in the Finite Static List (I_i) to another element in the Finite Static List (I_j), $i <> j$	E_{12} = [pending, hold] implies that the initial value was "pending" and changed to "hold". A change from "hold" to "hold" is not relevant
$b = 2^n * D_0(m) * D_1(m) \ldots * D_d(m)$ = number of bits needed to represent all possible (E_{ij}) state changes for all dependencies	$b = 2^4 * 3 * 2 * 2 = 192$. There are 192 possible state changes from one item in the list to another. There are 3 possible user authorizations. There are 2 possible types of applications. There are 2 possible ranges for loan amount. Number of bits needed is 192.
$E_{ij}(D_{0k}\ D_{1k} \ldots\ D_{dk})$ = A state change (E_{ij}) for a given set of dependencies.	Each of the 16 state changes must be replicated for each dependency.
S = A set of b bits, where each S_{ij} (D_{0k} $D_{1k} \ldots\ D_{dk}$)maps to each $E_{ij}(D_{0k}\ D_{1k} \ldots\ D_{dk})$	S contains 192 bits.
$P_{ij}(D_{0k}\ D_{1k} \ldots\ D_{dk})$ = Position of bit in S mapped for each $E_{ij}(D_{0k}\ D_{1k} \ldots\ D_{dk})$. $P_{ijk}(D_{0k}\ D_{1k} \ldots\ D_{dk}) = ((n*i + j) +$ $2^n * D_{0k})) +$ $(2^n * D_0(m) * D_{1k}) +$ $(2^n * D_0(m) * D_1(m) * D_{2k}) + \ldots +$ $(2^n * D_0(m) * D_1(m) * \ldots * D_{d-1}(m) * D_{dk})$ $P(D_{0k}\ D_{1k} \ldots\ D_{dk}) < b$	Changing from pending to hold with a user authorization of supervisor for an application that is parent initiated and a loan amount greater than $500: pending: i = 1; hold: j = 2, n = 4 supervisor: $D_{0k} = 1$, $D_0(m) = 3$ parent initiated: $D_{1k} = 1$; $D_1(m) = 2$ loan amount > $500; $D_{2k} = 1$, $D_2(m) = 2$ Bit position = $(4 * 1 + 2) +$ $(2^4 * 1) +$ $(2^4 * 3 * 1) +$ $(2^4 * 3 * 2 * 1)$ $= 166$
$S(P_{ij}(D_{0k}\ D_{1k} \ldots\ D_{dk}))$ = Value of bit in S at position $P_{ij}(D_{0k}\ D_{1k} \ldots\ D_{dk})$	
$S(P_{ij}(D_{0k}\ D_{1k} \ldots\ D_{dk})) = 0$ if $E_{ij}(D_{0k}\ D_{1k} \ldots\ D_{dk})$ is not a valid state change	If the value at the calculated bit position is a 0, it is not a valid state change.

$S(P_{ij}(D_{0k}\ D_{1k}\ldots\ D_{dk})) = 1$ if $E_{ij}(D_{0k}\ D_{1k}\ldots\ D_{dk})$ is a valid state change	If the value at the calculated bit position is a 1, it is a valid state change.
Condition-Action Rule: Given a status of I_i, a user authorization of D_{0k}, and application type of D_{1k}, and a loan amount of D_{2k}, determine whether a change to I_j is valid. If a change from I_i to I_j is not valid, do not allow the change. Evaluation: V = true if $S(P_{ij}(D_{0k}\ D_{1k}\ldots\ D_{dk})) = 1$ V = false if $S(P_{ij}(D_{0k}\ D_{1k}\ldots\ D_{dk})) = 0$	Given a user authorization of supervisor and a change from "pending" to "hold": Calculate the bit position: Initial Status (pending) = 1 Changed Status (hold) = 2 Supervisor authorization = 1 (of 3) Application type = 1 (of 2) Loan amount = 1 (of 2) Bit Position = $((4 * 1 + 2) +$ $(2^4 * 1) +$ $(2^4 * 3 * 1) +$ $(2^4 * 3 * 2 * 1)$ $= 166$ If value at bit position "166" = 1 it is a valid change If value at bit position "166" = 0 it is not valid

The key to understanding the changes made to the algorithm to accommodate the many dimensions of the business rules is that an offset must be calculated for each added dimension. If there are four items in the list, then there are 2^4 bits required to represent each valid state change. These 16 bits must be replicated for each of the three user authorizations, bringing the total to 48 bits. Each of these 48 bits must be replicated for both of the application types, which brings the total to 96. Finally, each of these 96 bits must be replicated for both of the loan amounts, which brings the total to 192 bits. These 192 bits represent the number of "if" statements that a programmer would have to generate to fully represent the business logic required. Furthermore, the 192 "if" statements would be nested the number of dimensions deep. Once again, the Java program given would need to be modified slightly to accommodate the additional dimensions by overloading some of the methods to additionally pass the user authorization, application type and loan amount to be used in the calculation of bit positions.

CONCLUSION

The dynamics of business logic in the life-cycle of software development deeply affect the development, testing and maintenance costs. It is through a well thought-out design of the business logic, where there is ease in programming, testing, and maintenance, that system development costs can be reduced.

The challenge with programming and consequently maintaining business logic arises from the concept of separating out rules for implementation outside of the language selected for program development. This immediately introduces a need to know which rules are to be implemented in this alternative mechanism and, by inference which specification mechanism to use for which rules. Designing the implementation of business rules should be a critical part of the requirements gathering and analysis.

It was not the purpose of this paper to identify the modeling of the business rules, but rather the underlying algorithm that supports the business rule. Future research in this area will be the further development of business rule engine algorithms. An extensive study that actually measures the success of such an implementation in development, testing, and maintenance should also be done.

REFERENCES

Belderrain, Cristina (2002) Message-Driven Beans and Encapsulated Business Rules. http//www.theserverside.com/resources/articles/ MessageDrivenBeansAndEncapsulatedBusinessRules/article.html.

Grosof, B. Labrou, Y. (1999) An Approach to using XML and Rule-based Content Language with an Agent Communication Language. IBM Technical Report RC21491.

Grosof, B., Labrou, Y. & Chan, H.Y (1999) Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. Proceedings of EC99.

Grosof, Benjamin, Rouvellou, Isabelle, Degenaro, Lou, Chan, Hoi, Rasmus, Kevin, Ehnebuske, Dave, McKee, Barbara (2000) Combining Different Business Rules Technologies: A Rationalization. Proceedings of the OOPSLA 2000 Workshop on Best-practices in Business Rule Design and Implementation.

Perkins, Alan (2000) Business Rules = Meta-Data. IEEE , 285-294.

Rosca, K., Daniela and D'Attilio, John (2001) Business Rules Specification, Enforcement and Distribution for Heterogeneous Environments. IEEE, 3-9.

Rosca, L., D., Greenspan, S., Wild, C., Reubenstein, H., Maly, K., Feblowitz, M. (1995) Application of a Decision Support Mechanism to the Business Rules Lifecycle. Proceedings of the KBSE95 Conference.

Rouvellou, Isabelle, Rasmus, Kevin, Ehnebuske, Dave, Degenaro, Lou, McKee, Barbara (2000). Extending Business Objects with Business Rules. IEEE.

Shao, G. Fu, Embury, S.M., Gray, W.A., Liu, X. (2001) A Framework for Business Rule Presentation