SSRG International Journal of Computer Science and Engineering
ISSN: 2348–8387 / https://doi.org/10.14445/23488387/IJCSE-V11I2P101

Volume 11 Issue 2, 1-14, February 2024
© 2024 Seventh Sense Research Group®

Original Article

Flexible Software Architecture for Validation of Change Requirements

Abhijeet R. Thakare[1], Atul O. Thakare[2], Omprakash W. Tembhurne[3], Soora Narasimha Reddy[4], Parag S. Deshpande[5]

[1]*MCA Department, Ramdeobaba College of Engineering and Management, Nagpur, Maharashtra, India.*
[2]*STME, SVKM's NMIMS University, Navi Mumbai, Maharashtra, India.*
[3]*CSE Department, MIT-ADT University, Pune, Maharashtra, India.*
[4]*CSE(Networks) department, Kakatiya Institute of Technology and Science, Warangal, Telangana, India.*
[5]*Visvesvaraya National Institute of Technology, South Ambazari Road, Nagpur, Maharashtra, India.*

[2]*Corresponding Author : aothakare@gmail.com*

Received: 01 January 2024 Revised: 03 February 2024 Accepted: 15 February 2024 Published: 28 February 2024

Abstract *- In a dynamic environment such as government organizations validation logic is changing because of the introduction of new laws or new procedures. Since government or public organizations may be legally scrutinized, such validations are important and have to be strictly enforced. In most organizations, this change is incorporated by changing software code or by changing the underlying structure of the database. Many times, this adds a lot of cost to the software, and sometimes, it may not be possible to modify the underlying structure due to the unavailability of domain experts and technical experts. The motivation behind the work is to be able to specify the domain requirements in the form of Entity and Attributes so that any user can specify the change in validation logic in a simple way. Suppose the validation logic is expressed in some programming language that operates on relational tables. In that case, the complexity of validation logic is quite large. Also, if validation logic changes, the developer must implement that logic by writing more lines of code in the corresponding programming language, which is very cumbersome and results in a waste of man-hours. A highly skilled developer will write 100 lines of code per day (Approximately 9 Hours). To avoid this, we have proposed expressing validation logic using high-level entity attributes, a natural data representation. We provided different operators in the language that are sufficient to express validation logic. The experimentation results prove that the proposed software system facilitates expressing the validation logic using high-level entity attributes and reduces the complexity and cost of the process of updating the validation logic.*

Keywords *- Entity and attributes, Validation logic, Domain requirements, Dynamic software design, Change request.*

1. Introduction

In today's highly dynamic environment, the only permanent thing is "change". Generally, complex enterprise systems meet with continuous evolutions. This change impact is adverse and cumulative. To get an idea of what "change" means in this context, consider an example of a government organization. Assume that a software system is designed using a traditional approach to maintain the details of the employees in the organization. This software system is designed keeping in perspective all the rules, regulations, and constraints specified concerning the employees in that organization. Government organizations are highly sensitive to new laws or new procedures. These new laws or new procedures are strictly enforced by government organizations in their software, occasionally for legal scrutinization purposes. New validation rules must be inserted to incorporate these new laws and procedures into the existing system. This leads to increased complexity and an increasingly dynamic nature of the software [1].

To implement these validations in the existing system, one has to update the underlying database structure, and the system's interface must be altered to be coherent with the underlying structure. This is sometimes much more burdensome than updating the database structure with the new validations. This means writing long and cumbersome database function code or queries. Just adding the database code does not satisfy the need, the database code needs to be tested to ensure that the code is correct because that database code should be tested and debugged properly. For performing this task, long man-hours are wasted unnecessarily. Another issue while enforcing new rules and regulations in the traditional system is that the database is designed by some person (or group of people). These people may not always be present; a new person may not understand the database design, which was designed formerly by those people. The new person would find enforcing these new rules and regulations extremely difficult or impossible. Another problem associated with the traditional approach is that every time the interface changes, the staff must be trained again and/or instructed to familiarize themselves with the updated interface. Hence, there is a need for a generalized model (interface) that will handle effectively the above-discussed scenarios. In the proposed research, we have focused on designing a novel high-level model based on entity and attributes. This interface will be able to accommodate dynamically changing validation logic.

1.1. Motivation

Various approaches have been proposed in the domain. These approaches bypass the models, which incorporate a high-level description in common vocabulary that only human experts can understand and are essentially consistent. These approaches are based on the underlying assumption that enforcing rules and regulations are known beforehand and are unlikely to change drastically in the foreseeable future. Under this assumption, it was possible to encode knowledge and implementation of the system with models as high-level specifications and generate platform-specific implementations. However, considering the current scenario, we observe that the rules and regulations are continually changing, and a need arises to update the system with the new rules and regulations. Thus, we needed to develop to accommodate the dynamically changing validation logic dynamically. In legacy approaches, the validation logic is expressed with respect to the relational tables, which have a high complexity of validation logic as it involves writing complex database language code. Hence, the need suggested that if we could specify the validation logic in the form of Entity and Attributes, then anybody would be able to manipulate the validation logic and would simplify the procedure of updating new rules and regulations in an existing system. Another concern behind the motivation of this topic was cost-effectiveness. The rules and regulations keep changing regularly; consequently, the validation logic must undergo radical changes. Many times, this adds a lot of cost to the software. Also, many times, it becomes impossible to implement the validation logic due to the lack of domain experts. This need triggered the motivation for the design of a system. Our system provides an interface to specify and implement the validation logic and can dynamically accommodate these changes. Writing long and cumbersome database code involves investing precious man-hours. An average programmer can code 1000 lines of code in 10 days on average. If we can save these precious man-hours, then those can be invested in something fruitful. Thus, simplifying the validation logic specification will save many man-hours invested in coding the long and cumbersome database code.

1.2. Contributions

- A novel design of an intermediate high-level interface that transforms high-level rules and regulations into a low-level database representation.
- A novel procedure to facilitate the specification and implementation of validation logic in the form of Entity and Attributes.
- Designing high-level functions and operators that assist in the specification of validation logic.
- Storing of validation logic for future demand by the system.

2. Related Work

To adapt to the drastically changing requirements, the software must be updated often with significant improvements made in a short amount of time. Demands severely test the software system's ability to change and improve quickly for new features and continuous enhancements. Software maintenance and software evolution are two different things. Bug fixes, small additions, and migration are generally referred to as software maintenance. Conversely, significant functional improvements and modifications are the main focus of software evolution. The research described in [2] distinguishes between two unique scenarios: evolution and maintenance. The term software maintenance has the objective of keeping the software error-free and up-to-date as per technological advancements. Software evolution has the objective of updating the software product as per the changing functional and non-functional requirements. Software evolution includes the perspectives of perfective and adaptive maintenance [3]. Therefore, the efforts required in software evolution amount to more than three-quarters of the maintenance activities. In most situations, evolution results from concurrent changes in several of the properties of one or more activities of an application [3]. Keeping software evolving while preserving the system's general stability and coherence is a difficult task for developers and maintainers. Work depicted in [4] defines software evolution as a feedback system with complex interaction and feedback control among software systems, development processes, and application environments. A system's environment (domain), requirements (experience), and implementation technologies (process) can all change over time, according to [5], which defines software evolution. Another definition of evolvability given by them is the capacity of a system to withstand modifications to its surroundings, specifications, and execution technologies.

A framework, Tropos4AS, mainly targeted for adaptive software systems, has been addressed in [6]. This framework is utilized successfully for requirement validation. In this framework, various approaches like Agent-oriented software engineering, goal-oriented requirements engineering, and BDI agent software platforms are integrated. The technique of validating big data streaming in an IoT environment has been addressed in [7]. Various formal techniques of performing requirement change management in software systems have been addressed in [8].

A requirement validation framework has been depicted in [9]. This framework, named Virtual Requirement Prototype (VRP), will decrease the cost and feedback duration of stakeholders. This activity is achieved by allowing Stakeholders to collaborate with a virtual prototype for certifying embedded software requirements. In the work done in [10], a method named CuRV is proposed. This method utilizes a mental framework. The mental framework technique is beneficial for examining customer's activity and mental condition. With the help of CuRV, a benchmark is established for validation. It is also useful to rank and ratify elicited requirements. CuRV is mainly useful for validating customer requirements. Work presented in [11] depicts a framework named AGG. With the help of AGG, Graph transformation is achieved. Validation of the software model is achieved with the help of the AGG framework. In the AGG framework, validation techniques evolved are based on a formal approach.

These techniques include phases such as Graph parsing, critical pain analysis, and consistency checking. The work addressed in [12] addresses a concurrency-centered framework. In the context of JPL's MDS Framework, this framework is helpful for real-time C++ semantic parallelization and validation. The temporal constraint network is a crucial component of executing mission planning and control architectures of the mission Data System Framework. For validation of semantic variants of the Temporal Constraint Network, a concurrency centered framework is beneficial. Various approaches are useful to validate software requirements. The work depicted in [13] proposes a technique for generating Natural Language text from process models. This technique is beneficial for validating user requirements effectively. In this complete process, requirements are initially transformed into a process model. In the work carried out in [14], a Web Usability Evaluation Process (WUEP) is proposed. This method in collaboration with Model-Driven Development, is utilized to perform usability evaluations in the early development stages of web development. WUEP has proven to be effective for performing empirical validation of requirements. A method developed in [15] explains a model-driven approach for validating spreadsheets. The task of co-evolution is also executed in this system. In the work [16], a prototype for validating typical features of critical software for nuclear power plants. This nuclear power plant is used for safety protection. Formal specification of language is useful for performing validation tasks. Work performed in [17], a system is proposed that automatically detects changes in Software Architecture. Various Graph Analysis Algorithms are applied to detect patterns in Software Architecture. The work proposed in [18] depicts the techniques of validating quality requirements for the software so that the cost of software validation is reduced. Work depicted in [19] proposes a toolkit that is utilized for keeping the Software Architecture up to the mark during software development. This toolkit (model) is based on language-independent meta-data. In the work [20], a malleable Software Architecture is proposed. Languages' syntax and semantics are represented in this Software Architecture. This syntax and semantics are meant for mobile agents. A-Line Information System Architecture (LISA) has been proposed in [21]. It offers a high level of scalability and flexibility for monitoring low-level operations and a high level of data. Work Presented in [22] depicts a Service Oriented Architecture based on an event. Aspects produced from SOA and EDA Architectures are dubbed, which results in extensive formalization of evaluation of Software Architecture Proposed.

3. Problem Definition

In the traditional approach for enterprise systems, a three-tier architecture is used, which compromises three layers: User Interface Layer, Business Logic Layer, and Database Layer, as described in Figure 1.

The problem with the above approach is that if the validation logic changes due to the introduction of new laws and procedures, the legacy software architecture seems incapable of coping with the need to specify and implement the validation logic dynamically. This is mostly because it involves modifying software code or updating the underlying structure. This is often a costly operation, and sometimes, it may be impossible to perform the modifications due to the lack of or unavailability of domain experts. To overcome this problem, we propose a high-level architectural design that will be flexible enough to deal with the manipulation of validating conditions in Entity and Attribute form.

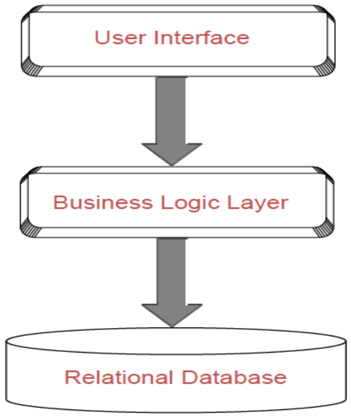


Fig. 1 Traditional Three-Tier enterprise system

3.1. Background of Validation Logic

Considering the overall system, it is a pretty obvious question: what is validation logic? The answer is simple. Whenever new rules and regulations are enforced in an organization, these rules need to be incorporated into the user database maintenance system. The validation logic can be simply perceived as translating these rules and regulations into the machine language (Figure 2). Validation provides confidence that the right product is being created. It further guarantees that the software being created (or modified) meets the needs of its stakeholders and that the laws are correctly and effectively implemented. It is a very important aspect in implementing the given set of rules and regulations. The rules and regulations are provided as company policies, which are supposed to be mapped onto the database in the form of validation logic so that whenever any act against the policies is performed, the validation logic would raise an exception, indicating that the corresponding rule(s) is violated. These validation logics are strictly scrutinized and hence require correctness and precision. Thus, if the database code is written in terms of relational tables directly, then it involves a lot more complexity in the validation logic because the rules specified are mostly pretty complex and descriptive in nature, thus resulting in hundreds of lines of code. But if this validation logic is specified in terms of some high-level operators and functions, then it will save a lot of time invested in writing those hundreds of lines of code, resulting in saving man-hours.

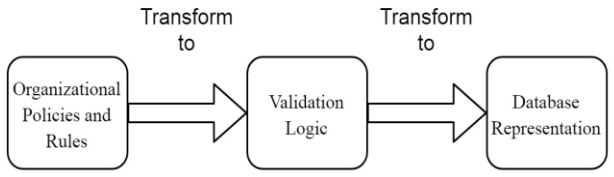


Fig. 2 Validation logic

3.2. Design Paradigm

The central theme of the architecture proposed in Figure 3 is to have a set of layers wherein adjoining layers will have a predefined and complete set of relationships. We have defined a layered architecture consisting of an intermediate layer and a high-level Entity-Attribute layer between the User Interface Layer and the Business Logic Layer. This intermediate high-level interface layer consists of a set of predefined operators sufficient to express the validation logic. By utilizing these operators, a naive, non-database user can specify and implement the validation logic.

This adds more flexibility to the software as the validation logic can be easily defined and manipulated. If the validation logic changes, the developer's effort is saved in changing the code compared to the traditional approach.

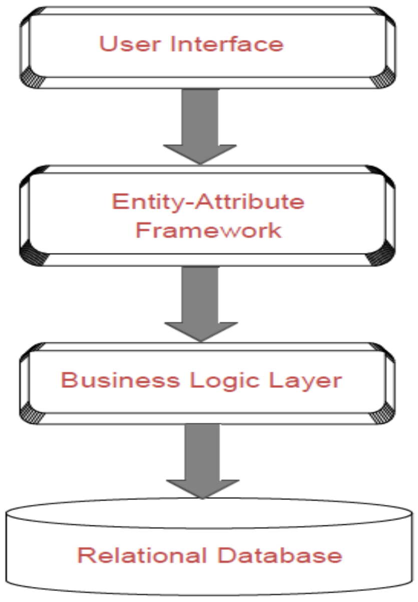


Fig. 3 Proposed model

As the business logic changes due to changes in client requirements, a developer doesn't require changing the entire code or changing the structure of the database. As Validation logic is part of a business logic layer, a developer will easily formulate the new validation logic using entity attributes, and his/her man efforts for changing the entire code or changing the structure of the database are drastically saved. The upcoming sections give a detailed description of the functionality of each layer.

3.2.1. User Interface (UI)

This layer is particularly associated with the validation logic capturing process. The user enters the validation logic through this interface. This interface design complies with dynamically realigning itself with the structure of the lower-level entity attribute framework. This means that any change taking place at the entity level is dynamically accommodated whenever the underlying entity level structure changes. We achieve this customization by capturing the entity details using the Meta-data stored in the database tables. The interface has a facility to enter user-defined validation logic names to identify the logic.

Further, we can go ahead and select the desired entity name from a drop-down list, which will eventually result in populating a drop-down list consisting of the attributes related to the selected entity. This list is dynamically populated with the relevant attributes of corresponding entities as and when the user makes a selection for an entity. Then, the user interface pops up the required number of input media to input the parameters for the corresponding entity-attribute pair. Again, this activity is event-driven and is dynamically manipulated as and when different selections are made. There is a custom provision made to input a numerical compare value instead of set operators in case relational operators are selected; otherwise, this input media can be left blank. The most important thing to highlight is that there are virtually no bounds or restrictions on how many conditions to insert in a validation logic. The user can input as many conditions as he wishes to add. This gives the user all the freedom to customize the validation logic according to the given scenario. This feature also helps achieve flexibility and preciseness regarding validation logic.

Another important point to be highlighted is that we can add multiple validation conditions under a single logic name. The Entity Attribute framework is designed to take care of this scenario, where the validation logic is independently evaluated, and each condition in that validation logic is evaluated separately. This feature gives a higher degree of preciseness and flexibility to specify the validation logic. The validation logic is associated with a truth value (i.e. true or false), and the evaluation is compared against the user-specified truth value. For simplicity, the implemented model considers only truth values for evaluation; however, in the future, numerical values can be considered for evaluation purposes.

In further sections, a detailed elaboration of specifying validation logic is provided. It describes how the design accomplishes the main goals of adaptability, ease of use, accuracy, etc.

3.2.2. Entity Attribute Framework

After the user enters the validation logic, it is passed on to this layer, which has several functionalities to perform. This layer primarily deals with the process of tokenization and parsing of the given validation logic. The validation logic is in a sequence of Entity Attributes pair form, which is raw data that must be processed. On the reception of validation logic, first of all, the logic is tokenized. The tokens are separated using ‘#’ as a delimiter. The obtained tokenized string is then parsed, and the appropriate meaning of each token is interpreted using a set of rules predefined in the grammar.

The grammar and the whole tokenization and parsing process are elaborated in detail in the upcoming sections. Briefly describing, the framework can be described as a language for processing the validation logic regarding Entity and Attributes. After parsing, validation logic proceeds for evaluation. For evaluation, we simply refer to the unique reference ID allotted to the validation logic and the parameter list essential for processing the validation logic. This whole functionality is performed by a special function that requires input from the logic reference ID and the parameter list. The user is also notified about the parameter list required for the evaluation. Analyzing the scenario from the user’s perspective, the user simply perceives that he defined the validation logic and provided the parameter list; the rest of the process is completely abstract to the user. What the user receives, in the end, is just the outcome of whether the validation logic is evaluated as true or false. This also tends to save a lot of man-hours that might have been invested without our system. This seems to be a big achievement in increasing the system's efficiency, as human error is much less because the programmer does not need to waste time coding and testing long and cumbersome database code. The primary contribution of this layer is incorporating simplicity, clarity, and abstraction in the language. This makes it easier for a naïve user to perceive the whole system as compared to the traditional systems, which tend to define all the functionalities at the table level.

3.2.3. Business Logic

This layer primarily interacts with the low-level database representation. In the Entity Attribute framework, a predefined set of operators operates on the parameters to generate either a set or a truth value as output. Each of the operators, defined in the business layer, is associated with a procedure. There is a one-to-one mapping between the operators and the database procedures. During the parsing phase, the database language code associated with the operator is fetched, and the procedure is called. Based on its functionality, the procedure generates either a truth value or a set as output. The output is given back to the Entity framework layer, which in turn gives it back to the user. The operators defined in the Entity-Attribute framework are sufficient to define any condition in the validation logic. The operators are designed to be as generic as possible. The operators are primarily of two types:1) Set 2) Relational. The union and intersection operators have the set return type. However, the rest of the operators have a Boolean return type.

3.2.4. Database Layer

For the overall system to be efficient, the database design has to be efficient. This layer depicts the relational database in the architecture. This layer has the lower-level database representation of the Entity-Attribute structure. The operators mentioned in the business logic layer are mapped to procedures in the database. Whenever a high-level operator is referred to in the validation logic context, the procedure from the database is invoked. When referring to the operator, the procedure code is fetched corresponding to that operator. For generalization, the attribute markers are used instead of actual parameters while storing them in the database. Also, an actual mapping from database language code to markers is stored in a separate set. This set is referred to whenever the procedure is called. So, the code with attribute markers is first fetched when the actual procedure needs to be called. Then, the corresponding markers for that particular code are replaced by the actual parameters. This phenomenon is analogous to the “pass by value" concept in an object-oriented language. This feature assists in the implementation of flexibility and generality in the system. As mentioned earlier, the database design has to be efficient for the system to be efficient. To achieve these, dozens of models were tried, tested, and failed until finally, we came up with this architecture model, providing us with features like flexibility, generality, simplicity, abstraction, etc. However, this arena has far more boundaries to be explored and captured. The upcoming sections now give a detailed description of every process.

4. Proposed Approach

4.1. Specifying a Validation Logic in UI

Specifying the validation logic is a much simpler task and easily perceivable by a naïve database user. The specification of the validation logic involves just making selections from a bunch of drop-down lists. To simplify this process, we realized that if the human involvement factor in decision-making is reduced, then we can achieve a higher degree of correctness in the system as the human involvement factor can't be eliminated at this stage, so as a step towards achieving this goal, as many drop-down lists possible are provided. This ensures that human errors in this context are minimized. In some areas, manual input is required; however, we are working on that part to reduce the human involvement factor further. Thus, all it takes to specify validation logic is making a selection using some drop-down lists. To specify validation logic, a user has to follow certain steps. As the validation specification relies primarily on the Entity-Attribute framework, a user needs to select an entity from the drop-down list. When selecting an entity, the relevant attributes associated with that entity are populated in the next drop-down list. The attributes and operators are populated dynamically. Then, the input fields are generated equal to the number of parameters required by that attribute for its evaluation.

Further, the parameters essential for processing the attribute must be entered. If required, the numerical compare value can also be specified depending on whether the attribute evaluates to a numeric value or set. This constitutes a single condition in the validation logic. As mentioned earlier, we can specify as many conditions as we wish. This flexibility is incorporated by adopting a unique naming scheme for all the HTML components that are present or dynamically generated. Also, the validation logic is associated with a truth value, which the user will specify. It is verified when the logic is evaluated, and the user is notified about the evaluation status.

All the operators and the functions are specified in simple representation so that any user can understand and specify the validation logic. Once the validation logic is specified, it is framed and stored in a special format using '#' as a delimiter. This step is mandatory for imparting ease in further processing of the validation condition. The primary advantage of implementing this flexibility is that the process of specifying validation logic is highly simplified.

We will elaborate on this scenario with the help of an example. Consider the validation logic condition specified as follows:

```
If (STUDENT.TOTAL_CREDITS < 50 AND

STUDENT.DEPARTMENT='CSE' AND

   STUDENT_COURSE.COURSE_NAME

  SUBSET COURSE.COURSE_NAME)

        RETURN TRUE;
ELSE

RETURN FALSE;
```

The condition in if-clause (highlighted) is what the user will be specified, and the overall structure is how the user will perceive the condition formulated. Here, "STUDENT" "STUDENT_COURSE" are Entities, and "TOTAL_CREDITS", "DEPARTMENT" and "COURSE_NAME" attributes. The validation logic is stored in the database as follows:

```
#STUDENT.TOTAL_CREDITS#<50#AND#STUD-
ENT.DEPARTMENT#='CSE'#AND

#STUDENT_COURSE.COURSE_NAME#SUBSET#COURSE.
COURSE_NAME#
```

Here, # is used as a separator for distinguishing between entity attributes, operators, connectors, and numeric values.

4.2. Processing of Validation Logic

The processing of validation logic is carried out in three particular phases. First of all, the validation logic string is tokenized. The tokenized string is parsed, referring to the language's grammar rules. Later, depending upon the return type of the operator, the result is returned and verified with the truth value provided by the user. The organization of the database also plays an important role in increasing the system's overall efficiency. The following sections provide a deep insight into how the validation logic is processed.

4.2.1. Database Organization

An entity may be defined as a thing that is recognized as being capable of independent existence and which can be uniquely identified. An attribute is a specification that defines a property of an object, element, or file. The database table schemas are a part of the Entity-Attribute framework and are referred to as an entity. The entities are represented in the form of sets for simplicity. All the attributes are associated with at least one parameter essential for the purpose of evaluation. A generic form of representation of an entity along with its associated attributes will correspond to something as depicted as follows:

S.T1 (p1), S.T2 (p2), S.T3 (p1, p2), S.T4 (p1, p2, p3, p4)

Where,

'S' → Entity

T1, T2, T3 and T4 →Attributes associated with the entity S.

p1, p2, p3, p4 → parameters associated with the corresponding attributes.

The entity-attribute representation suggests that parameter p1 is associated with attribute T1, p2 is associated with T2, p1, and p2 both are required for evaluation of T3, parameters p1, p2, p3, p4 all are mandatory to be specified for the evaluation of attribute T4.

Example - Consider an Entity Student with attributes as Roll_Number, Name, Gender, DOB, Age, total_credits, and subjects_count. The Entity student can be represented as:

Student. Roll_Number (roll_no),
Student. Name (roll_no), Student. Gender (roll_no), Student.DOB (roll_no), Student. Age (roll_no), Student. Total_credits (roll_no), Student. Subjects_count (level_no, roll_no)

Thus, it is evident that Name will require roll_no as a parameter to obtain its value compared against the user-specified value. Similarly, roll_no is a sufficient parameter to evaluate Roll_Number, Gender, DOB, Age, and total_credits. But, on the other hand, subjects_count require level_no and roll_no both of them for its evaluation.

4.2.2. Tokenization

A token is a set of one or more letters that have meaning together. Tokenization is the process of creating tokens from an input stream of text. The validation logic, after special amendments, is ready for the process of tokenization. Regardless of the quantity of words or inflectional ends, a word may have, a lexeme is a unit of lexical meaning. The defined token formats are sufficient for tokenizing all the lexemes specified in the validation logic. The Lexemes from the validation logic are identified based on the rules of the lexical analyzer. The lexemes in the validation logic are delimited with a special character (#) to facilitate the tokenisation process. Using '#' as a delimiter, we can easily distinguish between different types of lexemes. This approach of using a special character has a limitation; the special character used for delimitation cannot be used in the validation logic other than for delimitation purposes. A possible solution is using such a special character, which is highly unlikely to be used in daily life for that domain expertise. Since we considered the implementation for the college database system, and it seems '#' is unlikely to be used in this domain, '#' was a choice preferred for the delimiter.

There are four valid categories of tokens defined in our system:

Entity. Attribute

The first category of tokens identifies the entity and attribute in the lexeme. E.g. STUDENT.TOTAL_CREDITS. The attributes mentioned are of two types: i) Compound function with a return value (TOTAL_CREDITS), or ii) attribute type with no return value (e.g. ROLL_NUMBER). All the attributes require a particular number (at least one) of input parameters, and it returns either a Boolean value, Numerical value, or a Set. E.g. TOTAL_CREDITS takes roll_number as input parameter and returns the completed credits by the student whereas ROLL_NUMBER requires roll_no as input parameter and returns a boolean value. The set operators require input parameters and a return set of values to be fetched and processed if required.

Operator

This second category of tokens identifies the operator. The relational operators are generally associated with a value to be compared with. The operators are of two types: 1) Relational and 2) Set.

The Set operators are UNION, INTERSECTION, and SUBSET. The first two types return a set, whereas the former ones have a Boolean type. The Relational operators are <, >, <=, >=, ==. These operators require a compare value to be provided by the user to compare with the evaluated value of the corresponding attribute.

AND | OR

These tokens are meant to identify the fusing condition type, i.e. to identify whether the conditions in the if-clause are logically ANDed or logically ORed.

Operand

This token is used along with the relational operators, i.e. when relational operators are specified, the operand has to be specified. This type of token has literal values, which are alphanumeric in nature. This token represents the values that are specified in accordance to be compared with the return values of the attributes after evaluation.

For example, the values '50' and 'CSE' are categorized as the operand type of tokens.

E.g. consider the previously considered condition.
#STUDENT.TOTAL_CREDITS#<50#AND#ST-UDENT.DEPARTMENT#='CSE'#AND#
STUDENT_COURSE.COURSE_NAME#SUBSET#COU RSE.COURSE_NAME#

Now, the lexemes are classified based on their type of token, as shown in Table 1.

4.2.3. Parsing

Parsing is the act of examining a sequence of symbols, either in computer languages or natural languages, in accordance with the guidelines of a formal grammar. This phase constitutes one of the most important phases in the processing of the validation logic. After the validation logic is tokenized, the parsing phase comes into the picture. For interpreting each and every token, there is a set of production rules defined in the grammar.

Based on those rules, the evaluation of the validation logic takes place, and the result is returned to the user. The process of parsing is explained in detail below.

Table 1. Lexemes represented as tokens

Lexeme	Token Type
STUDENT.TOTAL_CREDITS	Entity.Attribute
<	Operator
50	Operand
AND	AND | OR
STUDENT.DEPARTMENT	Entity.Attribute
=	Operator
'CSE'	Operand
AND	AND | OR
STUDENT_COURSE.COURSE_NAME	Entity.Attribute
SUBSET	Operator
COURSE.COURSE_NAME	Entity.Attribute

4.2.4. Grammar

To understand the language framework, we need to understand grammar and its associated rules. The Context-Free Language is defined as a 5-tuple L= (V, ∑, P, N, S). The formal definition of the Context Free language is given below. The grammar and the production rules follow the language definition. An example of derivation of the validation logic accompanying grammar supports the grammar definition and verifies the production rules.

$$L = (V, \sum, P, N, S)$$

Where,
$V = (\sum \cup N)$ is a finite set of symbols called the vocabulary (or set of grammar symbols); $\sum \subseteq V$ is the set of terminal symbols (for short, terminals);

$S \in (V - \Sigma)$ is a designated symbol called the start symbol; $N = V - \Sigma$ is called the set of non-terminal symbols (for short, non-terminals); $P \subseteq (V - \Sigma) \times V^*$ is a finite set of productions (or rewrite rules, or rules).

The set of terminals and non-terminals is as follows:

Terminals (∑) → {< if >, < then >, < else >, < end >, < return >, < true >, < false >, <, >, <=, >=, ==, union, intersection, subset, <entity>, <attribute>}

Non-Terminals (N) → {S, C, X, O1, O2, E, A, D, K}

The formal set of production rules, P, is as given below:

S→<if>C<then>S<else>S<end>|<if>C<then>S <end>|<return><true>|<return><false>
C →C (X O1 X) C | C (X O2 K) C | and | or | €
X →E D A
K →[0-9] + | '[a-z A-Z]+'
O1 →union | intersection | subset
O2 →< | <= | > | >= | == | <>
E →<entity>
A →<attribute>
D → .
Note: Here,
< entity > corresponds to the Entity Name from Entity Attribute Framework.
e.g. STUDENT

< attribute > corresponds to the attribute names associated with the entity.
e.g. TOTAL_CREDITS

Since the list of Entities and attributes can be too long, we have abbreviated the description in this context.

E.g. consider the above-specified logic. First, we will verify whether the grammar actually derives the validation logic. The if-then-else grammar is adopted from

Compilers: principles, techniques, and tools, which ensure that the if-then-else grammar is free from hanging else. Hence, we will just focus on the grammar specified for the condition.

C → C (X O2 K) C
C → € (X O2 K) C
C → (EDA O2 K) C
C → (studentDA O2 K) C
C→ (student.A O2 K) C
C → (student.total_credits O2 K) C
C → (student.total_credits < K) C
C→ (student.total_credits < 50) C
C→ (student.total_credits < 50) C (X O2 K) C
C → (student.total_credits < 50) and (X O2 K) C
C → (student.total_credits < 50) and (EDA O2 K) C
C→ (student.total_credits < 50) and (studentDA O2 K) C
C→ (student.total_credits < 50) and (student.A O2 K) C
C→ (student.total_credits < 50) and (student.dept O2 K) C
C → (student.total_credits < 50) and (student.dept = K) C
C→ (student.total_credits < 50) and (student.dept = 'CSE') C
C→ (student.total_credits < 50) and (student.dept = 'CSE') C
C→ (student.total_credits < 50) and (student.dept = 'CSE') C (X O1 X) C
C→ (student.total_credits < 50) and (student.dept = 'CSE') and (X O1 X) C
C → (student.total_credits < 50) and (student.dept = 'CSE') and (EDA O1 X) C
C → (student.total_credits < 50) and (student.dept = 'CSE') and (student_courseDA O1 X) C
C → (student.total_credits < 50) and (student.dept = 'CSE') and (student_course.A O1 X) C
C→ (student.total_credits < 50) and (student.dept = 'CSE') and (student_course.course_name O1 X) C
C → (student.total_credits < 50) and (student.dept = 'CSE') and (student_course.course_name subset X) C
C → (student.total_credits < 50) and (student.dept = 'CSE') and (student_course.course_name subset EDA) C
C → (student.total_credits < 50) and (student.dept = 'CSE') and (student_course.course_name subset courseDA) C
C → (student.total_credits < 50) and (student.dept = 'CSE') and (student_course.course_name subset course.A) C
C → (student.total_credits < 50) and (student.dept ='CSE') and (student_course.course_name subset course.course_name) C
C → (student.total_credits < 50) and (student.dept = 'CSE') and (student_course.course_name subset course. course_name) €

The validation logic is derived using leftmost derivation, as shown in Figure 4. As evident from the derivation, the grammar is capable of parsing all types of validation logic identical to the type of logic in the example. Also, there is no restriction on the number of conditions inserted in the validation logic. Here, for simplicity, we considered three types of conditions to demonstrate the parsing of validation logic without any loss of generality. The validation logic would consist of conditions of a similar type. The only change will occur in the number of conditions and the type of operators and operands. We have considered just the parse tree for conditions since condition parsing is the only part considered during the storage and evaluation of the validation logic. Hence, the demonstration of parsing for specified conditions serves the purpose. That is the very reason we are considering 'C' as the start symbol and not 'S'. Using the grammar production rules, the parse tree s generated is shown in Figure 4.

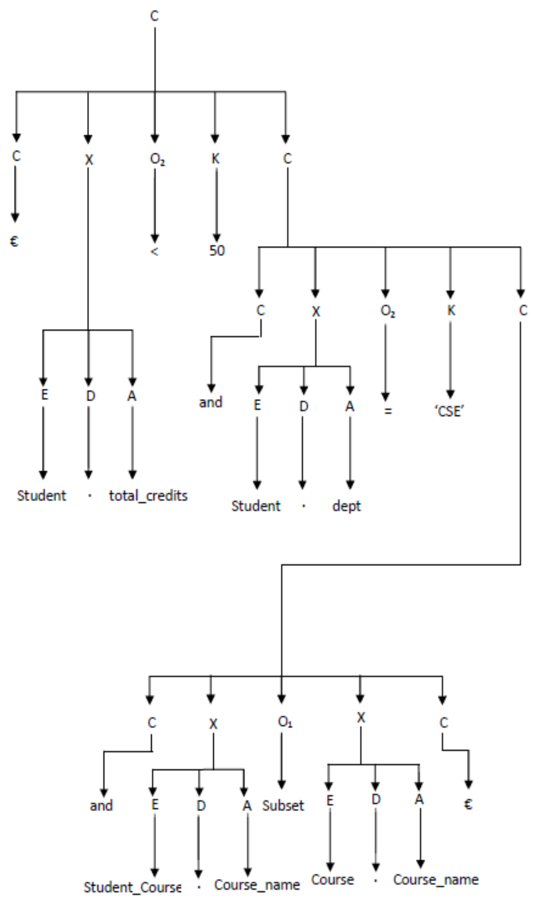


Fig. 4 Parse tree for the validation logic

5. Results and Result Analysis

5.1. Semantic Analysis or Evaluation

After the parsing of the validation logic is done, each condition in the validation logic is evaluated individually. For evaluation, the expression generated by the parse tree is considered. The expressions can be of the four types of tokens. Each token type is processed differently. This means the tokens of category I are processed to evaluate the database language code, if present, or simply the attribute value is compared to the value specified by the user in the category IV type of token, i.e. operand. We will provide a relational table representation of each entity required for processing and evaluating the specified validation logic. Depending on other pairs, some entity and attribute pairs are not mapped because those simply represent the columns in that particular relational table. This mapping is stored in a separate entity, 'parameter_attribute'. This entity can be described in Table 2. Here, each entity is individually mapped to a 'DBMS_LANGUAGE_CODE' whose functionality is predefined in the database.

It is the stored procedure that carries out the actual functionality of the attribute. For example, consider the entity-attribute pair "student.total_credits". For this particular pair, the database language code associated with it is eval_credits(#RN#, marks), which is a PL/SQL code predefined. So "eval_credits(#RN#, marks)" is the function prototype. This procedure takes the student's roll number as an input parameter and produces marks as the output of the type integer. Here, we observe that the student's roll number will be available only at the run time and not beforehand. So, for the process to be generalized, we make use of markers. A marker is a temporary replacement for the parameter and will be substituted by the actual parameter value during the time of evaluation. The markers are stored in a special format to facilitate the distinguishing markers from actual validation logic lexemes.

The markers have a generalized type as #marker_name#. Thus, "#RN#" is a marker associated with the entity total_credits. So, when the user specifies the roll number, the marker is replaced with the roll number, and then the call for the procedure is made. The mapping between the entity and its associated markers is stored in another entity, "ATTRIBUTE_MARKERS", which is described in Table 3. Every attribute has at least one parameter associated with it. That parameter is essential to fetch the values of the evaluation of entity attributes, which are then compared with user-specified values to return a Boolean result. So, "#RN#" is the marker associated with the attribute "total_credits". To get a deeper insight into the evaluation procedure, we will also consider the code for the stored procedure eval_credits(). We have tried and tested the code for MySQL and ORACLE SQL. Hence, we will consider MySQL code for demonstration purposes for the procedure, as shown in Figure 5.

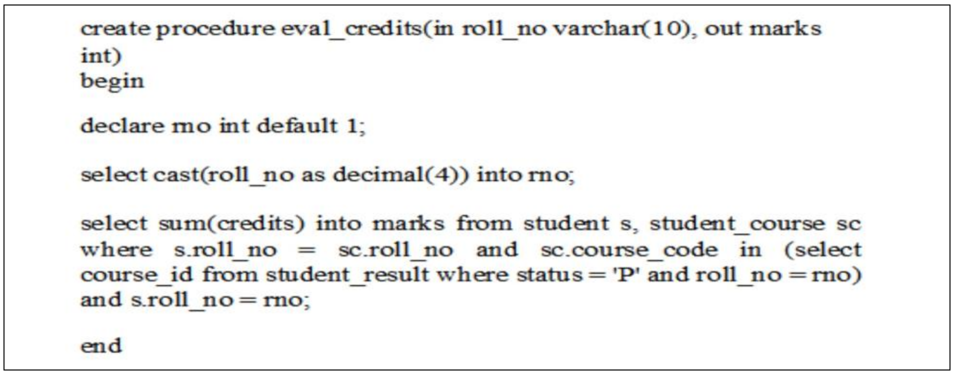
As evident from the code, the attribute total_credits returns the total credits scored by the roll number. Thus, this function simplifies the task of obtaining total_credits for a particular student. Had it been the case that this function would be absent, then one would have to write the whole query each time there would been a need to fetch the total credits scored by a student. This function simplifies the task of getting the total credits of the student and saves a lot of time required to be invested in writing the code. Thus, all we have to do is provide a particular roll number to the attribute and get the result immediately. These provide an easy and hassle-free way to perform the required functionality. This is a much simpler query than writing hundreds of lines of code.

Table 2. Representation of parameter_attribute entity in relational table format

Attribute_ id	entity_name	attribute_ name	Dbms _language_ code	attribute_ return _type	in_ param eters	out_ param eters	in_ out_ parame ters
1	student	total_ credits	eval_credits (#RN#,marks)	int	1	1	0
2	student	count_ subjects	count_subjects (#RN#, #LEVEL_REG_NO#, marks)	int	1	1	0

Table 3. Representation of Attribute_Markers entity in relational table format

marker_ id	attribute_ name	marker	marker_ data_type	marker_name
1	total_ credits	#RN#	int	roll_no
2	count_ subjects	#RN#	int	roll_no
3	count_ subjects	#LEVEL_REG_NO#	int	Level_ Registration_ Number



```
create procedure eval_credits(in roll_no varchar(10), out marks
int)
begin

declare rno int default 1;

select cast(roll_no as decimal(4)) into rno;

select sum(credits) into marks from student s, student_course sc
where s.roll_no = sc.roll_no and sc.course_code in (select
course_id from student_result where status = 'P' and roll_no = rno)
and s.roll_no = rno;

end
```

Fig. 5 Code for eval_credits () procedure

Considering a functionality consisting of hundreds and thousands of lines of code, if we can provide such functionality to ease off the coding process, we can save a lot of man-hours of programming invested in this coding. Procedures make use of dynamic SQL to dynamically construct queries at run time. After the procedure finishes its execution, the returned result is fetched and compared. A user-defined Java function, eval (), is employed to perform all these tasks. This function accepts the unique reference ID used to refer to stored validation logic, called logic_id, and the parameter list generated parallel during lexical analysis. Using the logic_id, the validation logic is extracted, and the above-described procedure is followed for each of the individual conditions from the validation logic. The validation logic is stored in the form of a string of characters in the database; hence, extracting all the entity-attribute pairs from the logic and individually processing them is a complicated task. The processing proceeds as follows. Firstly, the individual lexemes from the validation logic are delimited using '#' as a delimiter. By doing this, we obtain separate lexemes or tokens, to be precise. Secondly, as stated earlier, the tokens of four types are to be processed. To process the first category of tokens, we use '.'(dot) as a delimiter and separate the entity and the attribute from the category token. Then, we fetch the dbms_language_code from the parameter_attribute entity. Later, the markers from the attribute_markers entity for the corresponding attribute are fetched and substituted with the parameters from the parameter list. Then, the call to the stored procedure, if present, is made. The result is fetched and stored in a string variable, result, for evaluation. When tokens from other categories are encountered, different generalized methods are adopted to process them. The resultant string variable result is then fed to the javaScript Engine to evaluate the result. Here, we have another important case of first-category tokens where there is no stored procedure associated with the attribute, e.g. "student. dept == 'CSE'". The 'department' attribute associated with the 'student' entity does not need to be associated with any stored procedures. It can be processed independently. Another classification in the first category of tokens is the attributes with a SET return type. These types are identical to the previously mentioned type, but this token type needs to be processed individually and dependently due to the return type. Now, the parameter list is passed on to the specified validation logic. Assume we are evaluating the validation logic for roll number 2. So, the parameter list will contain roll number 2. Also, assume that the total_credits for roll number 2 is 46, but roll number 2 belongs to the 'EEE' department, and the courses that roll number 2 has enrolled for belong to the course names in the course entity. The resultant value of the returned result is the logical ANDing of the obtained truth values of the evaluated conditions. So (true && false &&true) evaluates to false, and hence true is returned to the user.

If similar validation conditions arise in the future, we simply need to fetch the validation logic from the database and provide the required parameter list for evaluation. Thus, in this case, to notify the user about the necessary and required parameters for evaluation, there is a custom provision to generate the parameter list. The user can then be prompted to input the desired parameters in the list and the validation logic can be evaluated.

5.2. Adding more Functionality to the Existing System

From earlier sections, we can say that even a naïve database user can easily specify the validation logic and enforce the rules and regulations using the proposed system. Also, the operators and attributes designed in the system are correct and sufficient to specify any validation logic. However, the need may arise in the future to add more functionality to the system.

As far as the introduction of new rules and regulations is concerned, we can add validation logic to the existing system. However, such a need to add more attributes corresponding to a particular entity may arise. So, associating new attributes is not difficult in this new system. As we know, the entity-attribute pair is mapped to the database language code in the parameter_attribute entity, and the corresponding markers, if any, are stored in the attribute_markers entity. So, to add a new attribute, we first need to write a pl/SQL code for the required functionality. Then, we need to store the exact prototype of the pl/SQL code under the dbms_language_code attribute in the parameter_attribute entity, associating the desired entity attribute pair with the required return type and parameters. Also, we need to store the markers necessary for the corresponding attribute in the attribute_markers entity. This is the case with entity attributes having database language codes associated with them. Even if we add new columns in an existing table, those attributes will be dynamically populated in the interface. In addition to this type of attribute, we do not need to perform any special activities. While populating the new attributes from the database, corresponding entities will be referred to, and appropriate attributes will be populated.

5.3. Integration and Testing

The system is developed using Java Server Pages (JSP) and Servlets as frontend and MySQL as backend. Due to the wide functionality and ease of access provided, JSP was adopted for designing and implementing the frontend. For testing purposes, MySQL is used. However, the system has also been tested for the Oracle SQL server. We tried and tested this model on two different databases viz., MySQL and ORACLE SQL. The results on both databases have been quite satisfactory.

In the previous chapter, we did mention both databases; however, for elaboration purposes, we chose MySQL. In MySQL testing, the overall dimensions of the database were small and limited. So, to make sure there are no performance issues, we also tested the system on our college's database, and the results have been quite good. The college's database is bigger in both dimensions and size. The system works fine on both databases and seems to have no performance issues.

6. Conclusion

Capturing the current need of the hour, we designed a flexible, dynamic, and simple system for capturing and implementing validation logic. Currently, in many government organizations, many new laws and procedures are added from time to time. These laws or procedures must be strictly enforced since the organizations are legally scrutinized. To accommodate these changes, the underlying structure often needs to be changed. This results in the addition of cost in the software or sometimes makes it impossible to incorporate these changes because of either poor database design or the unavailability of the domain expert. In the proposed system, we tried to reduce the human dependency factor in the process of specification and implementation of validation logic by providing an intermediate high-level entity attribute framework that translates the validation logic in terms of Entity Attributes to low-level database representation. The features and functionalities provided in the system seem to be sufficient and complete for specifying and implementing validation logic to the given date. By facilitating the specification of validation logic in the Entity Attribute form, we can save a lot of time invested in coding database code. Representation of validation logic in terms of Entity and Attributes also assists in imparting simplicity in the system as the user can now easily communicate with the system. The proposed system is flexible enough to accommodate any additions for functionality in the system without any serious modification in the underlying structure and interface.

Acknowledgments

All the authors are grateful to Dr. Parag. S. Deshpande, Professor & Head of Computer Science and Engineering Department, VNIT, Nagpur, for his valuable guidance and support for this work.

References

[1] Donald Firesmith, “Common Requirements Problems, Their Negative Consequences and the Industry Best Practices to Help Solve Them,” *Journal of Object Technology*, vol. 6, no. 1, pp. 17-33, 2007. [Google Scholar] [Publisher Link]

[2] B.P. Lientz, E.B. Swanson, and G.E. Tompkins, “Characteristics of Application Software Maintenance,” *Communications of the ACM*, vol. 21, no. 6, pp. 466-471, 1978. [CrossRef] [Google Scholar] [Publisher Link]

[3] Nazim H. Madhavji, Juan Fernandez-Ramil, and Dewayne Perry, *Software Evolution and Feedback: Theory and Practice*, John Wiley & Sons, 2006. [Google Scholar] [Publisher Link]

[4] M.M. Lehman et al., “Metrics and Laws of Software Evolution- The Nineties View,” *Proceedings Fourth International Software Metrics Symposium*, Albuquerque, NM, USA, pp. 20-32, 1997. [CrossRef] [Google Scholar] [Publisher Link]

[5] Selim Ciraci, and Pim van den Broek, “Evolvability as a Quality Attribute of Software Architectures,” *EVOL*, pp. 29-31, 2006. [Google Scholar] [Publisher Link]

[6] Mirko Morandini et al., “Engineering Requirements for Adaptive Systems,” *Requirements Engineering*, vol. 22, no. 1, pp. 77-103, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[7] Aparna Kumari et al., “Verification and Validation Techniques for Streaming Big Data Analytics in Internet of Things Environment,” *IET Networks*, vol. 8, no. 3, pp. 155-163, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[8] Shalinka Jayatilleke, and Richard Lai, “A Systematic Review of Requirements Change Management,” *Information and Software Technology*, vol. 93, pp. 163-185, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[9] Daniel Aceituna, Hyunsook Do, and Seok-Won Lee, "Interactive Requirements Validation for Reactive Systems Through Virtual Requirements Prototype," *2011 Model-Driven Requirements Engineering Workshop*, Trento, Italy, pp. 1-10, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[10] Youn Kyu Lee, Hoh Peter In, and Rick Kazman, "Customer Requirements Validation Method Based on Mental Models," *2014 21st Asia-Pacific Software Engineering Conference*, pp. 199-206, 2014, [CrossRef] [Google Scholar] [Publisher Link]

[11] Gabriele Taentzer, "AGG: A Graph Transformation Environment for Modeling and Validation of Software," *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pp. 446-453, 2003. [CrossRef] [Google Scholar] [Publisher Link]

[12] Damian Dechev et al., "Programming and Validation Techniques for Reliable Goal-driven Autonomic Software," *Autonomic Communication*, pp. 231-247, 2009. [Google Scholar] [Publisher Link]

[13] Henrik Leopold, Jan Mendling, and Artem Polyvyanyy, "Supporting Process Model Validation Through Natural Language Generation," *IEEE Transactions on Software Engineering*, vol. 40, no. 8, pp. 818-840, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[14] Adrian Fernandez, Silvia Abrahão, and Emilio Insfran, "Empirical Validation of a Usability Inspection Method for Model-Driven Web Development," *Journal of Systems and Software*, vol. 86, no. 1, pp. 161-186, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[15] Jácome Cunha et al., "Embedding, Evolution, and Validation of Model-Driven Spreadsheets," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 241-263, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[16] C.V. Ramamoorthy et al., "Application of a Methodology for the Development and Validation of Reliable Process Control Software," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 6, pp. 537-555, 1981. [CrossRef] [Google Scholar] [Publisher Link]

[17] Maayan Goldstein, and Itai Segall, "Automatic and Continuous Software Architecture Validation," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 59-68, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[18] Andrea Caracciolo, Mircea Filip Lungu, and Oscar Nierstrasz, "How Do Software Architects Specify and Validate Quality Requirements?," *European Conference on Software Architecture*, vol. 8627, pp. 374-389, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[19] Georg Buchgeher, and Rainer Weinreich, "Integrated Software Architecture Management and Validation," *2008 The Third International Conference on Software Engineering Advances*, Sliema, Malta, pp. 427-436, 2008. [CrossRef] [Google Scholar] [Publisher Link]

[20] Gabriel Ciobanu, and Călin Juravle, "Flexible Software Architecture and Language for Mobile Agents," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 6, pp. 559- 571, 2012. [CrossRef] [Google Scholar] [Publisher Link]

[21] Alfeed Theorin et al., "An Event-Driven Manufacturing Information System Architecture for Industry 4.0," *International Journal of Production Research*, vol. 55, no. 5, pp. 1297-1311, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[22] Sietse Overbeek, Marijn Janssen, and Patrick van Bommel, "Designing, Formalizing, and Evaluating a Flexible Architecture for Integrated Service Delivery: Combining Event-Driven and Service-Oriented Architectures," *Service Oriented Computing and Applications*, vol. 6, no. 3, pp. 167-188, 2012. [CrossRef] [Google Scholar] [Publisher Link]