

A Framework for Fast Service Verification and Query Execution for Boolean Service Rules

Soumi Chattopadhyay¹(✉), Saikat Dutta², and Ansuman Banerjee¹

¹ Indian Statistical Institute, Kolkata, India
soumi61@gmail.com

² Jadavpur University, Kolkata, India

Abstract. The problem of service rule verification has attracted some attention in recent years. In this paper, we consider service rules in simple Boolean logic and present a new method for business rule verification using simultaneous minimal support set computation. As we show here, the problem is similar in flavor to the problem of prime implicant generation of a given Boolean function which has alluded researchers for several decades and significant efforts in this direction have been reported in literature, with proposals of widely varying algorithms and data structures. In this paper, we revisit this problem in the context of business rules and present a new method that aids in rule verification and also in query execution at runtime. Our method builds on the classical binary decision diagram data structure for representing business rules and generates the test scenarios by a simple traversal algorithm. Experimental results on simulated benchmark rules show the efficacy of our approach.

1 Introduction

In recent years, the services ecosystem has seen a steady emergence of business rule management systems (BRMS) which allow enterprise architects to separate the concerns in an aspect-oriented way and easily define, manage, update and run the decision logic that directs enterprise applications in a Business rule engine (BRE) without needing to write code or change the business processes calling them [1–4]. This enables modern businesses to change their business rules dynamically in order to adapt to a rapidly changing business environment, and they contribute to agility in a service-oriented architecture paradigm by enabling reduced time to automate, easier change and maintenance for business rules.

In this services execution paradigm with business rules, the overall performance of the enterprise services delivered depends critically on the functional correctness and performance of the rule repository used for handling the business use cases. Bugs in rule encoding or implementation may create functional errors in service delivery, which may critically affect business performance. With web services and e-commerce being the order of the day like never before, ensuring correctness of a business rule set before deployment is becoming a critical mandate. In recent years, the problem of business rule verification has emerged as a problem of immense importance, due to failure scenarios of business cases that

have been reported by customers in interacting and working with several web services. This paper addresses this specific problem of business rule verification in a services ecosystem.

Business rules can be described in a declarative style [5,6] (using Declare models), or through a set of logical formulas (in Boolean/temporal logic [7] or their domain-specific variants). In this work, we have adopted a simple intuitive style for modeling business rules, using Boolean logic with predicates over arbitrary variables. Our motivation in this work is to address the problem of business rule verification, by which we can detect violations of business rules from the intended deployments or service level agreements. Indeed, this is of extreme importance, since the business rules form the critical component in a service delivery model. A number of research articles [8,9,14,15] have been reported in recent literature on this problem. These typically model business rules in either the declarative or the procedural style and attempt to use standard web service testing and formal verification methods to check for rule violations. While testing methods have their own limitations in terms of exhaustiveness of the scenarios covered, formal verification methods often fail to scale to the size and complexity of the business rule database that we are looking at. Moreover, most of these methods have restricted modeling styles for modeling business rules. In contrast, we attempt to present in this paper, a foundational framework for business rule verification by delineating the problem from the domain specifics, and instead adopting a simple intuitive classical logical style for expressing business rules. For the sake of simplicity and ease of illustration, we adopt Boolean logic with predicates for expressing business rules, and show that the problem of business rule verification can be cast as a simple instantiation of the classical prime implicant generation problem for Boolean functions. In this paper, we cast the problem of business rule verification as the task of extraction of test scenarios/queries for which the business rule is expected to evaluate to *true* (these are the ones in which a given business rule is supposed to trigger) and to *false* (scenarios where the rule is expected not to be exercised). Indeed, these constitute the scenarios that a business rule logic implementation need to be put against, since any violation of the expected outcomes on these scenarios is undesirable. The ability to extract and test against all the true and false scenarios gives us an added confidence of exhaustiveness akin to formal methods, while at the same time, makes our approach scalable and relevant in practice, since we do not suffer from any computational bottlenecks.

Computation of prime implicants for a Boolean function is one of the fundamental problems of Boolean algebra. Several approaches have been proposed in the literature that deal with this problem [10–13]. For business rule verification, we map the minimal support set generation problem as a simple variant of the classical prime implicant generation problem. Not only are we interested in extracting the scenarios in which a given business rule is expected to be triggered (which maps exactly to the prime implicant generation problem), we are also interested in extracting the use cases where the rule is expected to remain unexercised. This corresponds to another run of the prime implicant generation problem, for the negation of the given rule. In this work, we present a mechanism

by which we are able to unify the two tasks and address both the scenario generation problems in one pass. This is indeed an explicit novelty that we add on in this paper, and we contrast our approach through simulations on business rules, to show how much effort we save, in contrast to the two pass approach. Additionally, we address another interesting piece in this work. Not only are we interested in extracting the test scenarios, we also address the problem of computing the minimal cardinality valuations that can make a given rule true or false. This ensures that redundant scenarios are not used in our validation process and we indeed test a given business rule repository against a minimal relevant scenario set, while not compromising on the exhaustiveness of the verification process. The test scenarios synthesized by our approach are expected to be used by a rule logic implementation team to test their deployment models. Additionally, we can also take up these scenarios to check whether any of them leads to violations of the service level agreements. this serves as a key component in the rule verification process in a services deployment ecosystem, and we can use the rules to generate sample queries for the testing task. Another important aspect is the fact that this analysis can expedite the rule execution procedure. If we preprocess the rules and store the minimal support sets of the rule in an efficient manner, the queries can be answered at run time, without even executing the rule set, but using a simple look-up table. This in turn can expedite the query execution procedure as well. This paper has three key contributions, as outlined below:

- We address the problem of business rule verification, with rules expressed in extended Boolean logic and model it as an instantiation of the simultaneous test scenario generation problem.
- We discuss how we can expedite the process of run time query execution using a look-up table.
- We also present an innovative approach for simultaneously computing the exhaustive set of positive and negative test scenarios using a one-pass method, with the help of a novel data structure. This makes our proposition scalable and exhaustive and usable in practice.

2 Motivation for This Work

In this section, we illustrate the motivation of our work using a simple example. We consider an example business decision rule \mathcal{R} in an online shopping framework defined as follows:

*Example 1. **Rule:*** If the brand is ADIDAS (p_1) and *any* of the following conditions is true:

- It is Christmas time (p_2)
- For other times of the year (\bar{p}_2), if the customer is a valuable ADIDAS customer (p_3)
- On purchase from the old stock (p_4)
- On purchase above \$ 150 (p_5)

Then announce 10 % discount on every shopping from ADIDAS.

For convenience of notation and simplicity of expression, we introduce the predicates p_1, p_2, \dots, p_5 in the ruleset above that express the different conditions and use them in the following discussion throughout this paper. The triggering condition of the above rule can be expressed as the following Boolean expression.

$$\mathcal{A} = p_1 \cdot (p_2 + \bar{p}_2 \cdot p_3 + p_4 + p_5)$$

A simple analysis of the antecedents of the rules reveals the following scenarios where the rule is always true: {the brand is ADIDAS, this is Christmas time}, {the brand is ADIDAS, the customer is valuable}, {the brand is ADIDAS, purchase is done from the old stock}, {the brand is ADIDAS, purchase is more than \$ 150}. If we notice these scenarios carefully, we can observe that all the conditions are not present in every scenario, but still we can decide the rule as *true* and can actuate the corresponding consequent (discount announcement) of the rule. In fact these scenarios are minimum in cardinality, i.e., if we remove any of the conditions from inside any of the scenarios (comma separated list), the rule cannot be decided for a truth value. Similar is the situation for scenarios for which the rule is always false: {the brand is not ADIDAS}, {it is not Christmas time, the customer is not a valuable ADIDAS customer, the purchase is not done from the old stock, the purchase is less than or equal to \$ 150}.

If we can pre-process the antecedents of a given rule as discussed above, we can generate the sample queries to guide the functional testing of the rule set. A lot of approaches exist in literature which attempt to find these scenarios, using methods based on, test generation using support sets for Boolean function. Karnaugh map (K-map) [10] is one such popular method which can generate support sets, but it does not scale with the number of variables. Another popular method is the one proposed by Quine McCluskey [11]. The main disadvantage of this approach is, we cannot use this approach for a large number of inputs, because all minterms of a function need to be stored simultaneously in memory and to generate both the scenarios discussed above, we need to execute this method twice. Our proposed method, on the other hand, is able to generate both the scenarios simultaneously.

Another important use case, as already discussed is that our preprocessing proposal helps to expedite the run time service execution against incoming queries as well. When the actual query (valuation of the variables appearing in the rule) comes at run time, without even executing the rule set, we can decide which rules are true by simply using a look-up table, which can be used to store the support set valuations for which the rule evaluates to true or false. This greatly expedites the query evaluation process, since we do not need to explicitly evaluate a query for every input scenario. The support sets can guide us here as well. We explain in the following section, the technical details.

3 Detailed Methodology

In this section, we formally discuss our methodology. Figure 1 shows the different components of our method. The input of our method is the set of service rules.

The rule has two parts an *If* part and the *Then* part. In this paper, we assume that the *If* part is expressed in simple Boolean logic. *Then* part is the consequent part of the rule, it is actuated when the *If* part of the rule is evaluated as true. From here onwards, in our discussions, we consider the *If* part of the rule and we show how we can compute the supporting and refuting scenarios for the *If* part simultaneously that can expedite rule execution. The preprocessor preprocesses the rules and generate exhaustive set of positive and negative test cases which are stored in a lookup table. On one hand, this lookup table is used to verify the services, on the other side it is used for faster query execution at run time. We discuss both of these later in this section.

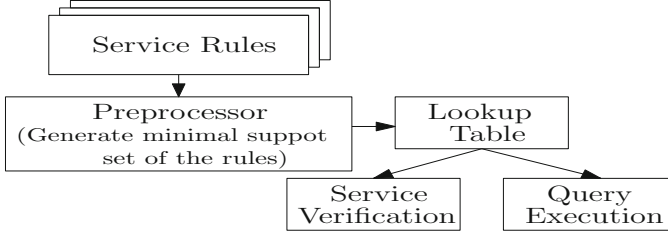


Fig. 1. Component diagram of our method

Before going into the details of the algorithm, we define a few terminologies that are necessary to develop our algorithm. We begin with some background concepts.

Definition 1. Support Set: A set $\mathcal{U} = \{(u_1, x_1), (u_2, x_2), \dots, (u_l, x_l)\}$ is said to be a support set of a service rule Φ defined over a set of Boolean propositions $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$, where $x_i \in \{0, 1\}$ and $u_i \in \mathcal{P} \forall i = 1, 2, \dots, l$, if Φ evaluates to either 0 or 1, when $u_1 = x_1, u_2 = x_2 \dots, u_l = x_l$. ■

Example 2. Consider a service rule $\Phi = p_1 \cdot (p_2 + p_3 + p_4 \cdot p_5 + p_6 \cdot p_7 \cdot p_8) \cdot \{(p_1, 1), (p_2, 1), (p_4, 1), (p_5, 1)\}$ is a support set of Φ , since Φ evaluates to 1 on the assignment $p_1 = 1, p_2 = 1, p_4 = 1, p_5 = 1$. ■

A support set of a service rule Φ for which Φ evaluates to true is called a *positive support set* and a support set for which Φ evaluates to false is called a *negative support set*.

Definition 2. Minimal Support Set: For a service rule Φ , a support set \mathcal{U} is said to be a prime, if no proper subset of \mathcal{U} is a support set of Φ . ■

It is worth noting that we redefine the classical notion of minimal support set by including the minimal support set with its valuation.

Example 3. For the function given in Example 2, $\{(p_1, 1), (p_2, 1)\}$ is a minimal support set of Φ . However, $\{(p_1, 1), (p_2, 1), (p_4, 1), (p_5, 1)\}$ is not a minimal support set of Φ , since it is a superset of $\{(p_1, 1), (p_2, 1)\}$, which is a minimal support set of Φ . ■

We now define the co-factor of a service rule.

Definition 3. Co-factor: The positive (negative) co-factor of a service rule Φ defined over a set of Boolean propositions $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$ with respect to a proposition $p_i \in \mathcal{P}$ is obtained by substituting 1 (true) / 0 (false) in Φ . ■

The positive co-factor, denoted by Φ_{p_i} is obtained by substituting the variable p_i with 1 in Φ , i.e., $\Phi_{p_i} = \Phi(p_1, p_2, \dots, p_i = 1, \dots, p_n)$. Similarly, the negative co-factor, denoted as $\Phi_{\bar{p}_i}$ is obtained as $\Phi(p_1, p_2, \dots, p_i = 0, \dots, p_n)$. The co-factors are independent of the proposition p_i with respect to which they are computed. We now define the concept of decomposition of a service rule. This follows as a straightforward application of Shannon's expansion [16].

Definition 4. Service Rule Decomposition: The decomposition of a service rule Φ with respect to a proposition $p \in \mathcal{P}$ is obtained as:

$$\Phi = p.\Phi_p + \bar{p}.\Phi_{\bar{p}},$$

where Φ_p and $\Phi_{\bar{p}}$ are respectively the positive and negative cofactors of Φ with respect to p . ■

Service rule decomposition can be extended to multiple propositions as well. From the definition of a minimal support set, it is trivial to observe that the co-factor of a service rule Φ with respect to its minimal support sets is always constant, i.e., either 0 or 1. Essentially, if $\{(p_1, 1), (p_2, 0)\}$ is a minimal support set of a service rule Φ , then $\Phi_{p_1 \bar{p}_2} = \text{constant}$, i.e. 1 or 0. We now define a few concepts which are important for our methodology and serve as the foundation.

Definition 5. Strong Proposition: A proposition u is said to be a strong proposition with respect to a minimal support set \mathcal{U} of a service rule Φ , where, $\mathcal{U} = \{(u_1, x_1), (u_2, x_2), \dots, (u_k, x_k)\}$ and $(u, x) \in \mathcal{U}$, if $\Phi_{p=\bar{x}}$ is independent of $\{u_1, u_2, \dots, u_k\}$. ■

Example 4. Consider the service rule $\Phi = p_1.p_2 + p_3.(p_4.p_5 + p_6) \cdot \{(p_1, 1), (p_2, 1)\}$ is a positive minimal support set of Φ . p_1 is a *strong* proposition with respect to $\{p_1, p_2\}$, since, $\Phi_{p_1=0}$ (i.e. $\Phi_{\bar{p}_1}$) is independent of p_2 . On the other hand, $\{(p_1, 0), (p_3, 0)\}$ is a negative minimal support set of Φ . It is easy to see that $\Phi_{p_1=1}$ (i.e. Φ_{p_1}) is *not* independent of p_3 . Therefore, p_1 is not a strong proposition with respect to $\{(p_1, 0), (p_3, 0)\}$. ■

Definition 6. Strong Minimal Support Set: A minimal support set \mathcal{U} is said to be a strong minimal support set with respect to a service rule Φ , if all the propositions appearing in \mathcal{U} are strong. ■

Example 5. Consider $\Phi = p_1.p_2 + p_3.p_4$. Here, $\{(p_1, 1), (p_2, 1)\}$ is a strong minimal support set, since, $\Phi_{\bar{p}_1}$ is independent of p_2 and $\Phi_{\bar{p}_2}$ is independent of p_1 . ■

It is intuitively obvious that a minimal support set with cardinality 1 is trivially a strong minimal support set.

Definition 7. Derivative: The derivative of a service rule Φ with respect to a proposition p_i is defined as the exclusive-or of the positive and negative co-factors of Φ with respect to the proposition p_i , i.e., $\partial\Phi/\partial p_i = \Phi_{p_i} \oplus \Phi_{\bar{p}_i}$. ■

In order to determine whether a proposition is strong with respect to a function Φ , we use the concept of the derivative.

Definition 8. Critical Proposition: A proposition p is said to be a critical proposition with respect to a service rule Φ , if the derivative of Φ with respect to the proposition p is 1, i.e., $\partial\Phi/\partial p = 1$. ■

3.1 Algorithm for Minimal Support Set Generation

In this section, we discuss our algorithm for simultaneous generation of positive and negative minimal support sets for a given service rule. Consider a service rule Φ defined over a set of Boolean propositions $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$. A naive approach for computing the minimal support sets of a service rule Φ is as follows: We choose a combination $\mathcal{C} = \{(p_1, x_1), (p_2, x_2), \dots, (p_k, x_k)\}$, where, $x_i \in \{0, 1\}$, $\forall i \in \{1, 2, \dots, k\}$. We compute the co-factor of Φ with respect to \mathcal{C} and check whether Φ evaluates to a constant. We start with a combination of cardinality 1 and then gradually increase the cardinality. While doing so, we keep track of the combinations for which Φ evaluates to a constant and do not consider any super set of such a combination. For each proposition p_i in the service rule Φ , there are three possibilities, either $(p_i, 0) \in \mathcal{C}$ or $(p_i, 1) \in \mathcal{C}$ or p_i does not belong to the propositions appearing in \mathcal{C} . Therefore, this naive procedure will lead us to explore all $3^n - 1$ combinations, which is very inefficient. We propose below our modified approach that does the same job in a more efficient way. We use the reduced ordered binary decision diagram (ROBDD) [17] data structure representation for Boolean functions as the backbone of our method. Algorithm 1 presents our approach for generating the minimal support sets of a service rule. It is an iterative algorithm. We gradually build up a tree \mathcal{T} to find out the minimal support sets. Each step of the algorithm is discussed below.

We now explain the detail of each step below. We start with a dummy node \mathcal{S} and incrementally iteratively build the complete minimal support set tree.

Simplify: In this step, we simplify the service rule Φ by removing all the critical propositions of the function. We identify all the critical propositions with respect to Φ and substitute them by 0 in Φ . The simplified function is used in the later steps of the algorithm.

Consider the propositions u_1, u_2, \dots, u_l such that $\partial\Phi/\partial u_j = 1$, for $j = 1, 2, \dots, l$. Then Φ can be written as, $\Phi = u_1 \oplus u_2 \oplus \dots \oplus u_l \oplus \Phi_{\bar{u}_1.\bar{u}_2.\dots.\bar{u}_l}$. Consider $\Phi_1 = \Phi_{\bar{u}_1.\bar{u}_2.\dots.\bar{u}_l}$. We find the minimal support sets of Φ_1 and then combine them with the 2^l combinations of u_1, u_2, \dots, u_l to obtain the minimal support sets of Φ . Lemma 1 expresses the correctness of this step.

Lemma 1. If $\partial\Phi/\partial p = 1$, every minimal support set of Φ has $(p, 0)$ or $(p, 1)$. ■

Algorithm 1. *GenerateMinimalSupportSet*

-
- 1: *Input:* A service rule Φ ; *Output:* Set of minimal support sets
 - 2: Construct the ROBDD(\mathcal{B}) for Φ ;
 - 3: **Simplify** Φ ; Create a start node \mathcal{S} ;
 - 4: **while** Φ is not constant **do**
 - 5: **Generate** partial minimal support set and construct \mathcal{T} ;
 - 6: **Substitute and simplify** Φ ;
 - 7: **end while**
 - 8: **Back propagate** from leaf node to start node;
-

Proof. Using Shannon's expansion we have, $\Phi = p.\Phi_p + \bar{p}.\Phi_{\bar{p}}$. Since $\partial\Phi/\partial p = 1$, $\Phi_p = \bar{\Phi}_{\bar{p}}$. $\Phi = p.\bar{\Phi}_{\bar{p}} + \bar{p}.\Phi_{\bar{p}} = p \oplus \Phi_{\bar{p}}$. $\Phi_{\bar{p}}$ is independent of p . The positive minimal support set of Φ = positive minimal support set of $\Phi_{\bar{p}} \cup \{(p, 0)\}$, since positive minimal support set of $\Phi_{\bar{p}}$ makes it 1 and if we substitute p by 0 then we get 0 and XOR of 0 and 1 is 1. As a result we get the positive minimal support set of Φ . Similarly positive minimal support set of Φ = negative minimal support set of $\Phi_{\bar{p}} \cup \{(p, 1)\}$. On the other hand, negative minimal support set of Φ = positive minimal support set of $\Phi_{\bar{p}} \cup \{(p, 1)\}$, or negative minimal support set of $\Phi_{\bar{p}} \cup \{(p, 0)\}$. Hence, every minimal support set of Φ contains either $(p, 1)$ or $(p, 0)$. ■

The advantage of this step is, if a service rule Φ contains only those propositions, with respect to which the derivative of the function is 1, we do not need to proceed further. Each combination of the propositions gives a minimal support set of Φ in such a situation.

Example 6. Consider $\Phi = p_1 \oplus (p_2 + p_3)$. Since, $\partial\Phi/\partial p_1 = 1$, the minimal support sets of $(p_2 + p_3)$ are $\{\{(p_2 = 1)\}, \{(p_3 = 1)\}, \{(p_2 = 0), (p_3 = 0)\}\}$. The minimal support sets of Φ are $\{\{(p_1 = 1, p_2 = 1)\}, \{(p_1 = 0, p_2 = 1)\}, \{(p_1 = 1), (p_3 = 1)\}, \{(p_1 = 0, p_3 = 1)\}, \{(p_1 = 1, p_2 = 0, p_3 = 0)\}, \{(p_1 = 0, p_2 = 0, p_3 = 0)\}\}$.

Partial Minimal Support Set Generation: In each iteration of the algorithm, we modify the original service rule Φ . In the next step, we discuss the modification of Φ . In this step, we generate the minimal support set of the modified service rule, we call it its partial minimal support set. Later, when we back trace through the minimal support set tree \mathcal{T} , we modify the partial ones to get the minimal support set of Φ . Algorithm 2 shows the formal procedure to generate a partial minimal support set. There may be multiple partial minimal support sets at this step. However, we generate one and proceed to the next step.

Once we obtain a strong minimal support set of the modified service rule, we create a leaf node \mathcal{L} and an intermediate node \mathcal{I} of the tree in the same level, i.e., both of them have the same parent. The parent node of the first level is the start node \mathcal{S} . The leaf node \mathcal{L} contains two parameters: the strong minimal support set of the modified service rule and the corresponding functional value,

Algorithm 2. *GeneratePartialMinimalSupportSet*

```

1: Input : ROBDD for the modified function  $\Phi_1$ 
2: Output : A strong minimal support set
3: for each proposition  $p$  associated with  $\Phi_1$  do
4:   loop
5:     Find a positive minterm  $\mathcal{M}_1$  associated with  $p$  from  $\mathcal{B}$ ;
6:     Find a minimal support set  $\mathcal{P}_1$  from  $\mathcal{M}_1$ ;
7:     if  $p$  belongs to the proposition set of  $\mathcal{P}_1$ , then break;
8:   end loop
9:   if  $\mathcal{P}_1$  is a strong minimal support set, then return  $\mathcal{P}_1$ ;
10:  loop
11:    Find a negative minterm  $\mathcal{M}_2$  associated with  $p$  from  $\mathcal{B}$ ;
12:    Find a minimal support set  $\mathcal{P}_2$  from  $\mathcal{M}_2$ ;
13:    if  $p$  belongs to the proposition set of  $\mathcal{P}_2$ , then break;
14:  end loop
15:  if  $\mathcal{P}_2$  is a strong minimal support set, then return  $\mathcal{P}_2$ ;
16: end for
17: return NULL;

```

i.e., 1 if it is a positive minimal support set and 0 if it is a negative one. The intermediate node also has two entries.

- A set of Boolean propositions assigned with a value, combined using OR.
- An assigned value, either 0 or 1.

Algorithm 3 shows the formal procedure to create an intermediate node. Initially, we pass \mathcal{S} in Algorithm 3 as the parent node. The intermediate node(s), obtained from the current level, is (are) going to be the parent node(s) in the next level. The interpretation of an intermediate node is as follows:

- If the assigned value of an intermediate node is 0, it implies that the content of the intermediate node is combined with a positive minimal support set generated at any of the levels lower than the one to which the intermediate node belongs to.
- Similarly, if the assigned value of an intermediate node is 1, it implies the content of the intermediate node is combined with a negative minimal support set generated at any of the levels lower than the one to which the intermediate node belongs to.

If we do not obtain any strong minimal support set in this step, instead of creating a single intermediate node, we create two intermediate nodes. The first intermediate node contains $(p, 0)$ in its first field and X (unknown) in its second field while the second intermediate node contains $(p, 1)$ in its first field and X in its second field, where p is a proposition associated with Φ_1 . If the assigned value of an intermediate node is X , it implies that whether the content of the intermediate node is combined with a minimal support set generated at any of the levels lower than the one to which the intermediate node belongs to is

Algorithm 3. *CreateIntermediateNode*

```

1: Input : Strong minimal support set of  $\Phi_1$ :  $\mathcal{U}$ , parent Node  $\mathcal{P}$ 
2: if  $\mathcal{U}$  is NULL then ▷ No strong minimal support set exists
3:   Create two intermediate nodes  $\mathcal{I}_1$  and  $\mathcal{I}_2$ ;
4:   Choose a proposition  $p$  associated with  $\Phi_1$ ; ▷ preferably first node of the
     ROBDD of  $\Phi_1$ 
5:   Assign  $(p, 0)$  to the first field of  $\mathcal{I}_1$  and  $X$  to the second field;
6:   Assign  $(p, 1)$  to the first field of  $\mathcal{I}_2$  and  $X$  to the second field.
7:   Add two edges from  $\mathcal{P}$  to  $\mathcal{I}_1$  and from  $\mathcal{P}$  to  $\mathcal{I}_2$ ;
8: else
9:   Create an intermediate node  $\mathcal{I}$ ;
10:  Assign tuples  $(u_i, \bar{x}_i)$  corresponding to each tuple  $(u_i, x_i) \in \mathcal{U}$  combined using
     OR, to the first entry of the intermediate node.
11:  if  $\mathcal{U}$  is a positive minimal support set then
12:    Assign 1 to the  $2^{nd}$  field of the intermediate node;
13:  else if  $\mathcal{U}$  is a negative minimal support set then
14:    Assign 0 to the  $2^{nd}$  field of the intermediate node;
15:  end if
16:  Add an edge from  $\mathcal{P}$  to  $\mathcal{I}$ ;
17: end if

```

decided after verification, i.e., we have to verify whether the minimal support set generated at any of the levels lower than the one to which the intermediate node belongs to, is a minimal support set of this level or not. This step is justified by the following lemmas.

Lemma 2. *If $\mathcal{U} = \{(u_1, x_1), (u_2, x_2), \dots, (u_k, x_k)\}$ is a positive (negative) minimal support set of Φ , where, $x_i \in \{0, 1\}, i = 1, 2, \dots, k$, every negative (positive) minimal support set contains at least one (u_i, \bar{x}_i) , such that, $(u_i, x_i) \in \mathcal{U}$. ■*

Proof. To prove this lemma, we need to prove two things: The set of propositions in a negative minimal support set \mathcal{U}_1 of Φ has a non empty intersection with the set of propositions in \mathcal{U} and every negative minimal support set contains at least one (u, \bar{x}) , such that, $(u, x) \in \mathcal{U}$. We prove both the claims by contradiction. Let us first consider a negative minimal support set of Φ , $\mathcal{U}_1 = \{(w_1, y_1), (w_2, y_2), \dots, (w_l, y_l)\}$, where, $y_i \in \{0, 1\}$ for $i = 1, 2, \dots, l$ and $w_i \notin \{u_1, u_2, \dots, u_k\}, \forall i \in \{1, 2, \dots, k\}$. Therefore the truth table of Φ contains at least one row satisfying \mathcal{U} and \mathcal{U}_1 simultaneously, since the intersection of the proposition set in \mathcal{U} and \mathcal{U}_1 is empty. This contradicts the fact that $w_i \notin \{u_1, u_2, \dots, u_k\}, \forall i \in \{1, 2, \dots, k\}$. So, the proposition set of \mathcal{U}_1 contains at-least one $u_i \in \mathcal{U}$, for $i = 1, 2, \dots, k$. Now we consider the fact \mathcal{U}_1 does not contain any (u, \bar{x}) , such that, $(u, x) \in \mathcal{U}$. Hence, we assume \mathcal{U}_1 contains $\{(u_{i_1}, x_{i_1}), (u_{i_2}, x_{i_2}), \dots, (u_{i_l}, x_{i_l})\} \subseteq \mathcal{U}$. Now if we merge the elements of \mathcal{U} and \mathcal{U}_1 , it remains a support set, say \mathcal{W} . It is easy to see that, \mathcal{U} and \mathcal{U}_1 are both subsets of \mathcal{W} , but \mathcal{U} is a positive minimal support set of Φ and \mathcal{U}_1 is a negative support set of Φ . This contradicts our assumption. ■

Lemma 3. *If $\mathcal{U} = \{(u_1, x_1), (u_2, x_2), \dots, (u_k, x_k)\}$ is a strong positive (negative) minimal support set of Φ , where $x_i \in \{1, 0\}, i = 1, 2, \dots, k$, every negative (positive) minimal support set of Φ contains exactly one (u_i, \bar{x}_i) , where $(u_i, x_i) \in \mathcal{U}$. Also no other positive (negative) minimal support set of Φ contains any (u_i, x_i) or (u_i, \bar{x}_i) where $(u_i, x_i) \in \mathcal{U}$. ■*

Proof. We present the proof for the positive minimal support set case and the proof for the negative minimal support set is similar. The proof is as follows: Since $\mathcal{U} = \{(u_1, x_1), (u_2, x_2), \dots, (u_k, x_k)\}$ is a strong positive minimal support set of Φ , $\Phi_{u_i=\bar{x}_i}$ is independent of $\{u_1, u_2, \dots, u_{i-1}, u_{i+1}, \dots, u_k\}$. If another minimal support set contains (u_i, \bar{x}_i) , it cannot contain any proposition from $\{u_1, u_2, \dots, u_{i-1}, u_{i+1}, \dots, u_k\}$.

Therefore a minimal support set can contain at most one $(u_i, \bar{x}_i) \in \mathcal{U}$. From Lemma 2, it follows that each negative minimal support set contains at least one element from the positive minimal support set with opposite polarity. Therefore, every negative minimal support set contains exactly one (u_i, \bar{x}_i) for $i = 1, 2, \dots, k$.

Now we prove the second claim, i.e., no other positive minimal support set of Φ contains any (u_i, x_i) or (u_i, \bar{x}_i) where $(u_i, x_i) \in \mathcal{U}$.

Case 1: Consider a positive minimal support set $\mathcal{W} = (w_1, y_1), (w_2, y_2), \dots, (w_l, y_l)$ of Φ such that $(u_i, \bar{x}_i) \in \mathcal{U}$ and $(u_i, \bar{x}_i) \in \mathcal{W}$. $w_j \notin \{u_1, u_2, \dots, u_{i-1}, u_{i+1}, \dots, u_k\}$ for $j = 1, 2, \dots, l$, since \mathcal{U} is a strong minimal support set. Consider a tuple $(u_j, x_j) \in \mathcal{U}$ and $u_j \neq u_i$. Clearly, when we substitute u_j by \bar{x}_j , the function Φ does not become independent of u_i which contradicts the fact that \mathcal{U} is a strong minimal support set.

Case 2: Consider a positive minimal support set $\mathcal{W} = (w_1, y_1), (w_2, y_2), \dots, (w_l, y_l)$ of Φ such that $(u_i, x_i) \in \mathcal{U}$ and $(u_i, x_i) \in \mathcal{W}$. Then there exists, at-least one $(u_j, x_j) \in \mathcal{U}$ such that $u_j \neq u_i$ and $(u_j, x_j) \notin \mathcal{W}$, otherwise \mathcal{W} would not be a minimal support set. If $(u_j, \bar{x}_j) \in \mathcal{W}$, then this case would be similar to Case 1. We can conclude that u_j does not belong to the proposition set of \mathcal{W} . Therefore, when we substitute any u_j by \bar{x}_j , the function Φ does not become independent of u_i , which again contradicts the fact that \mathcal{U} is a strong minimal support set. ■

Substitution and Simplification of Φ : Once the intermediate node is created, we modify the service rule, which we have now, say Φ_1 for the sub tree rooted at the intermediate node. Consider a strong minimal support set $\{\mathcal{U}\}$ generated in the current iteration. Assume, $\mathcal{U} = \{(u_1, x_1), (u_2, x_2), \dots, (u_k, x_k)\}$. Let us consider a tuple (u_i, x_i) from \mathcal{U} . We substitute $(u_i = \bar{x}_i)$ in Φ_1 and simplify the function to get the modified Φ_1 .

Back Tracing Through the Minimal Support Set Tree T : Once the entire minimal support set tree is created using the steps discussed above, we back trace through this tree to compute the minimal support sets. All the minimal support sets of Φ_1 generated in this step, have to be modified in order to get the minimal

support set of Φ . For that, we need to back trace through the intermediate nodes till the start node is obtained. If we consider the positive minimal support set of Φ_1 , while traversing backward we consider only the intermediate nodes which have assigned value 0. On the other hand, if the minimal support set of Φ_1 is negative, we similarly consider only the intermediate nodes with assigned value 1. If the assigned value of an intermediate node is X , we verify whether the minimal support set is going to be combined with the content of the intermediate node, in order to be a minimal support set of the service rule corresponding to the step to which the intermediate node belongs to.

3.2 A Complete Example

In this subsection, we explain the working of Algorithm 1 using an example. Consider the following rule:

Rule: If the brand is ADIDAS (p_1) and *any* of the following conditions is true:

- It is Christmas time (p_2)
- The customer is a new ADIDAS customer (p_3) and he purchases above \$ 150 (p_4)
- The customer is a frequent ADIDAS customer (p_5) and he purchases from new stock (\bar{p}_6)
- The customer is a infrequent ADIDAS customer (\bar{p}_5) and he purchases from old stock (p_6)
- The customer is a frequent ADIDAS customer (p_5) and he purchases above \$ 100 (p_7)

Then announce 10 % discount on every shopping from ADIDAS.

The *If* part of the rule can be expressed as:

$$\Phi = p_1 \cdot (p_2 + p_3 \cdot p_4 + p_5 \cdot \bar{p}_6 + \bar{p}_5 \cdot p_6 + p_5 \cdot p_7)$$

We wish to find all the minimal support sets (positive and negative) for Φ .

[**Step 1:**] (Simplify Φ): We construct the ROBDD for Φ . We find the derivative of Φ with respect to all the propositions appearing in Φ . We create a start node \mathcal{S} of the minimal support set tree \mathcal{T} .

[**Step 2:**] The aim of this step is to find a strong minimal support set of Φ_1 and create one/two intermediate node(s) of \mathcal{T} as needed. The iteration is started from this step. Let us assume the first proposition we consider here is p_1 . It is easy to observe that $(p_1, 0)$ is a strong minimal support set of Φ_1 . We find a positive and a negative minterm from the ROBDD of Φ_1 which contain p_1 and from these two minterms, we find a positive and a negative minimal support set containing p_1 as described in Algorithm 2. This has already been substantiated in Lemma 3. Eventually, we get a strong minimal support set $(p_1, 0)$. We create a leaf node \mathcal{L} containing $(p_1, 0)$ in its first field and 0 in its second field. We also create an intermediate node \mathcal{I} containing $(p_1, 1)$ in its first field and 0 in its second field as shown in Fig. 2.

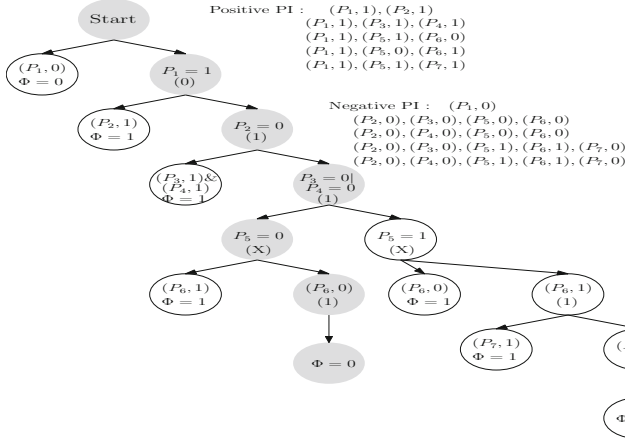


Fig. 2. Minimal support set computation for $\Phi_1 = p_1 \cdot (p_2 + p_3 \cdot p_4 + p_5 \cdot \bar{p}_6 + \bar{p}_5 \cdot p_6 + p_5 \cdot p_7)$

[**Step 3:**] We substitute p_1 by 1 and modify Φ_1 . Now the modified function is $\Phi_1^{(1)} = p_2 + p_3 \cdot p_4 + p_5 \cdot \bar{p}_6 + \bar{p}_5 \cdot p_6 + p_5 \cdot p_7$. For the sake of simplicity of illustration, we have used superscripts to differentiate the functions generated at each iteration and differentiate from the ones generated in other iterations. In this way, we create all the intermediate nodes corresponding to a strong minimal support set as shown in Fig. 2. In iteration 4, we have the modified function as $\Phi_1^{(2)} = p_5 \cdot \bar{p}_6 + \bar{p}_5 \cdot p_6 + p_5 \cdot p_7$. As we can see, the algorithm fails to find a strong minimal support set. Therefore, we create two intermediate nodes in this step \mathcal{I}_1 and \mathcal{I}_2 . Assume the proposition which we consider in this step is p_5 . Hence, \mathcal{I}_1 contains $(p_5, 0)$ in its first field and X in its second field. Similarly, \mathcal{I}_2 contains $(p_5, 1)$ in its first field and X in its second field. The modified function for the subtree corresponding to \mathcal{I}_1 is $\Phi_1^{(3)} = p_6$ and the modified function for the subtree corresponding to \mathcal{I}_2 is $\Phi_1^{(4)} = \bar{p}_6 + p_7$. We again start to find the minimal support sets of $\Phi_1^{(3)}$ and $\Phi_1^{(4)}$ according to our algorithm. Once the tree is constructed fully, we start back tracing in order to find the minimal support sets of $\Phi_1 = p_1 \cdot (p_2 + p_3 \cdot p_4 + p_5 \cdot \bar{p}_6 + \bar{p}_5 \cdot p_6 + p_5 \cdot p_7)$.

[**Step 4:**] (Back Propagation): Consider the shaded path in Fig. 2. The leaf node indicates that we are going to construct a negative minimal support set, since the leaf node contains $\Phi = 0$. The assigned value of the previous intermediate node is 1, hence we consider its first field $(p_6, 0)$. The next intermediate node contains an assigned value X , therefore we have to verify whether we consider $(p_5, 0)$ as follows. The service rule corresponding to this level is $\Phi_1^{(2)} = p_5 \cdot \bar{p}_6 + \bar{p}_5 \cdot p_6 + p_5 \cdot p_7$. It is easy to see that we have to consider $(p_5, 0)$ in order to get a minimal support set, since $\Phi_1^{(2)}$ does not evaluate to constant, if we substitute $(p_6, 0)$ in $\Phi_1^{(2)}$. The next intermediate node contains 1 in its second field. Therefore we have to consider its first field. Here we get two minimal support sets, one combining $(p_3, 0)$ and another combining $(p_4, 0)$. Similarly we have to consider

Table 1. Results of our implementation (MSS stands for minimal support sets)

Rules	Input variables	Positive MSS	Negative MSS	Quine McCluskey (ms)	Our method (ms)	CUDD (ms)
1	30	9936	13	Timeout	264.319	497.556
2	21	9	1296	Timeout	30.132	82.645
3	4	4	3	0.922	0.226	0.046
4	4	4	2	0.879	0.132	0.039
5	18	9	512	Timeout	6.814	23.527
6	36	64	87205	Timeout	30445	113824
7	3	1	3	0.333	0.111	0.033
8	3	1	3	0.331	0.079	0.029
9	3	1	3	0.33	0.1	0.042
10	2	1	2	0.04	0.066	0.021
11	4	1	4	4.029	0.09	0.041
12	7	5	3	4127.66	0.163	0.131
13	3	3	1	0.33	0.088	0.034
14	3	3	1	0.443	0.083	0.03
15	3	3	1	0.331	0.085	0.032
16	3	2	2	0.147	0.108	0.031
17	29	282	2196	Timeout	1219.72	1257.8
18	36	4434	11209	Timeout	155199	158713
19	35	1905	10228	Timeout	27078	62563.8
20	32	802	4692	Timeout	5333.16	9520.12

the next intermediate node as well. We do not need to consider the intermediate node containing $(p_1, 1)$ since its second field contains 0. The next node is the start node itself and therefore the back propagation terminates. In this step we get two negative minimal support sets: $\{(p_2, 0), (p_3, 0), (p_5, 0), (p_6, 0)\}$ and $\{(p_2, 0), (p_4, 0), (p_5, 0), (p_6, 0)\}$. The final step is to combine either $(p_0, 0)$ or $(p_0, 1)$ with each minimal support set in order to get the minimal support sets of Φ .

4 Implementation and Results

We implemented our methodology in C++. We ran our experiments on manually created random service rules of varying sizes and complexity. On one side, we obtain the positive and negative minimal support sets related to any proposition, as required by Algorithm 2 by iterating over the paths of our data structure. We also implemented the Quine McCluskey algorithm in C++ to provide a contrast of its efficiency against ours, which is shown in Table 1. The CUDD [18] Boolean function manipulation package provides several programmable interfaces for minimal support set generation, and we contrast our approach against the CUDD routine as well in the same table. Table 1 presents the experimental results. Columns 5, 6 and 7 show the time taken by Quine McCluskey, our method and CUDD Implementation respectively. As evident from Table 1, our

method takes much less time as compared to Quine McCluskey on all the benchmarks. Quine McCluskey is inherently nonscalable, hence we obtained timeouts on some of the larger cases. We also have comparative performance improvement over the CUDD API as evident on some of the cases. As explained earlier, we employed both the Quine McCluskey and CUDD API on the original function and its negation and recorded the combined times. As it can be seen, the CUDD implementation fails to generate all minimal support sets in many of the cases.

5 Conclusion and Future Work

In this paper, we address the problem of business rule verification and query execution, with rules expressed in extended Boolean logic. We discuss how we can expedite the run time execution using a look-up table and finally we present an innovative approach for simultaneously computing the exhaustive set of positive and negative test scenarios using a one-pass method, with the help of a novel data structure. This makes our proposition scalable and exhaustive. Experimental results on simulated benchmark shows the efficacy of our proposal. As evident from the results, our method efficiently computes the positive and negative scenarios as well. We are currently working on real business decision rules to see how our method works when put into real practice. Also we are experimenting on query evaluation using our method.

References

1. Paschke, A., Teymourian, K., AG Corporate Semantic Web: Rule based business process execution with BPEL+. In: I-SEMANTICS (2009)
2. Rosenberg, F., Dustdar, S.: Business rules integration in bpel-a service-oriented approach. In: E-Commerce Technology, CEC 2005 (2005)
3. Weigand, H., van den Heuvel, W.-J., Hiel, M.: Rule-based service composition and service-oriented business rule management. In: ReMoD (2008)
4. Paschke, A., Kozlenkov, A.: A rule-based middleware for business process execution. In: Multikonferenz Wirtschaftsinformatik (2008)
5. JRULEENGINE. <http://jruleengine.sourceforge.net/>
6. DROOLS. <http://www.drools.org/>
7. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. IEEE (1977)
8. Deutsch, A., et al.: Automatic verification of data-centric business processes. In: Proceedings of the 12th International Conference on Database Theory. ACM (2009)
9. Shi, Y.-L., et al.: TLA based customization and verification mechanism of business process for SaaS. *Jisuanji Xuebao (Chin. J. Comput.)* **33**(11), 2055–2067 (2010)
10. Karnaugh, M.: The map method for synthesis of combinational logic circuits. *Am. Inst. Electr. Eng. Part I: Trans. Comm. Electron.* **72**(5), 593–599 (1953)
11. McCluskey, E.: Minimization of Boolean function. *J. Bell Syst. Tech.* **35**, 1417–1444 (1956)
12. Coudert, O.: Two-level logic minimization: an overview. *Integr. VLSI J.* **17**(2), 97–140 (1994)

13. Ron, R.: An SE-tree-based prime implicant generation algorithm. *Ann. Math. Artif. Intell.* **11**, 351–365 (1994)
14. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based verification of web service compositions. In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, 6–10 October 2003, pp. 152–161 (2003). doi:[10.1109/ASE.2003.1240303](https://doi.org/10.1109/ASE.2003.1240303)
15. Zhu, Y., Gao, H.: A novel approach to generate the property for web service verification from threat-driven model. *Appl. Math.* **8**(2), 657–664 (2014)
16. Hachtel, G.D., Somenzi, F.: *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Dordrecht (2000). ISBN:0792397460
17. Huth, M., Ryan, M.: Binary decision diagrams. In: *Logic in Computer Science: Modelling and Reasoning About Systems*, Chap. VI, pp. 316–374 (2000)
18. CUDD: CU Decision Diagram Package Release 2.5.0. <http://vlsi.colorado.edu/~fabio/CUDD/cuddAllDet.html>

Advances in Services Computing

9th Asia-Pacific Services Computing Conference, APSCC

2015, Bangkok, Thailand, December 7-9, 2015,

Proceedings

Yao, L.; Xie, X.; Zhang, Q.; Yang, L.T.; Zomaya, A.Y.; Jin,

H. (Eds.)

2015, XIV, 312 p. 109 illus. in color., Softcover

ISBN: 978-3-319-26978-8