Abstract—As one type of machine-learning model, a “decision-tree ensemble model” (DTEM) is represented by a set of decision trees. A DTEM is mainly known to be valid for structured data; however, like other machine-learning models, it is difficult to train so that it returns the correct output value for any input value. Accordingly, when a DTEM is used in regard to a system that requires reliability, it is important to comprehensively detect input values that lead to malfunctions of a system (failures) during development and take appropriate measures. One conceivable solution is to install an input filter that controls the input to the DTEM, and to use separate software to process input values that may lead to failures. To develop the input filter, it is necessary to specify the filtering condition of the input value that leads to the malfunction of the system. Given that necessity, in this paper, we propose a method for formally verifying a DTEM and, according to the result of the verification, if an input value leading to a failure is found, extracting the range in which such an input value exists. The proposed method can comprehensively extract the range in which the input value leading to the failure exists; therefore, by creating an input filter based on that range, it is possible to prevent the failure occurring in the system. In this paper, the algorithm of the proposed method is described, and the results of a case study using a dataset of house prices are presented. On the basis of those results, the feasibility of the proposed method is demonstrated, and its scalability is evaluated.

I. INTRODUCTION

Recently, software developed by machine learning has been used in various systems. Deep learning using deep neural networks (DNNs) is widely used for predicting and classifying image data, audio data [1] [2], and so on. For structured data, ensemble learning methods using decision trees, such as random forests [3] and gradient-boosting decision trees [4], are also effective [5] [6] [7] [13] [8] [9] [10] [11].

A decision-tree ensemble model (DTEM) is represented as a set of decision trees. The prediction result output from a DTEM is calculated as the sum or average value of the scores associated with the leaves of the decision trees. The DTEM is expected to be generalized, namely, to return the appropriate output value even if it is given an input value not included in the training data.

However, in general, it is difficult to train a DTEM to return the appropriate output value for every input value, that is, a DTEM returns an inappropriate output value with a certain probability. Therefore, in particular, when a DTEM is used in a mission-critical system, whose behavior significantly affects business and society, it is important to comprehensively detect input values that lead to system failures during development, and take appropriate measures. As countermeasures, retraining or additional training of the DTEM are possible means; however, for the reason mentioned above, it is difficult to completely eliminate the possibility of failures occurring. Accordingly, as a practical measure, it is possible to create an input filter to control the input value to the DTEM and to use separate software to process the filtered input values that leads to failures (Fig. 1). The implementation of the separate software is arbitrary. For example, the input value might be rejected to be processed.

A policing function [12] is also useful to prevent failures. It is separated from the DTEM, and checks the output value of the DTEM against a certain property at runtime. If the output value does not meet the property, that is, the output value leads to failures, the policing function controls the output value. However, when comparing the input filter and the policing function, the input filter is more efficient because the input filter detects and controls failures before the DTEM runs.

To create the input filter, it is necessary to specify which input values should be filtered as the filtering condition. In this paper, we therefore propose a method to formally verify whether a DTEM meets certain property and, if that properties is not satisfied, extracts the range (part of the input-value space) in which all input values violating the property are included. By setting the range extracted by the proposed method as the filtering condition of the input filter, it is possible to prevent the failure of the system due to the DTEM. The feasibility of the proposed method is evaluated by showing a case study in which the proposed method was applied experimentally. Moreover, the scalability of the proposed method is evaluated by changing the dimension of the input values of the DTEM, the number of decision trees constituting the DTEM, and the maximum value of the depth.
of those decision trees. Hereafter, input values that violate the property are referred to as “violating input values.” And the range in which the violating input value exists is called the “violation range.” It should be noted that not all input values included in the violation range will be violating input values.

As for the proposed method, the decision trees that compose the DTEM are encoded to a formula, and the formula is verified by using a satisfiability modulo theories (SMT) solver. Although this approach has mainly been applied to DNNs [13] [15] [16] [17] [18] [19], to the authors’ knowledge, no case in which this approach is applied to a DTEM has been reported. Given that situation, the first contribution of this paper is to demonstrate that our approach, namely, logically encoding a machine-learning model and verifying the model by solving the resulting a formula with an SMT solver, is also applicable to a DTEM.

When the DTEM violates a property, as a result of the verification, an example of a violating input value (and its corresponding output value)—called a “counterexample”—is obtained. As a naive way to get the filtering condition, all violating input values are detected by repeating the verification. If an input value matches any violating input values, it is filtered. However, a large number of similar violating input values may exist around a certain violating input value. In particular, if the input value is represented by a multidimensional vector whose elements are numeric variables (not categorical variables), input values obtained by slightly changing the values of some elements of the violating input value are also likely to violate the property. In this case, since the verification is repeated as many times as the number of the violating input values, detecting all the violating input values is not practical in terms of calculation time.

Targeting a DTEM whose input values are multi-dimensional vectors whose elements are numeric variables, the proposed method extracts the violation range by searching around the origin at which a violating input value was first detected and gradually expanding the search range until the violating input value is not detected. With this method, the violating input values around the origin can be kept within the range. However, input values that do not violate the property are also included in the range. Even though they do not violate the property, they are also filtered as well as the violating input values. Thus, it is desirable to prevent as many input values that do not violate the property as possible from being included in the violation range. Therefore, as for the proposed method, the extracted violation range is divided into a number of smaller ranges. Then, for each divided range, whether a violating input value exists within the range is checked. As a result, it is possible to narrow down the original violation range. As for the second contribution of this paper, targeting a DTEM taking multi-dimensional vectors of numerical variables as input values, we propose a method to extract the violation range and narrow it down. Moreover, the feasibility and scalability of the proposed method are evaluated through a case study.

The rest of this paper is organized as follows. In Section II a decision tree and a DTEM are formally defined. In Section III the proposed method is overviewed. In Section IV among the procedures that compose the proposed method, the procedure for verifying the DTEM is explained. In Section V the procedure for extracting the violation range on the basis of the verification result is explained. In Section VI the procedure for narrowing down by dividing the extracted violation range is explained. In Section VII the feasibility and scalability of the proposed method are evaluated through a case study using a data set of house prices. In Section VIII the usefulness and applicability of the proposed method are discussed. In Section IX related work is described, and in Section X the conclusions drawn from this study are presented.

II. PRELIMINARIES

For any DTEM $M$, variables representing input and output values of $M$ are given as $x$ and $y$, respectively. It is assumed that $x$ is represented by a vector of length $s \geq 2$ such that $[x[0], \ldots, x[k], \ldots, x[s-1]]$. As for $M$ targeted in this paper, since multi-dimensional vectors with numeric variables as elements are assumed as input values, $x[0], \ldots, x[k], \ldots, x[s-1]$ represents each numeric variable. And the domain of $M$ is represented as $X$. First, in Section II-A the decision trees that compose $M$ are formally defined. Next, in Section II-B $M$ is formally defined.

A. Decision Tree

A decision tree represents a procedure for determining the class to which a given input value belongs [20] [21]. A decision tree consists of decision nodes, edges, and leaf nodes. The expression of input value $x$ is associated with the decision node (Fig. 2). In this paper, the expression of the decision node is called an “attribute test.” Each decision node has multiple child nodes, and each child node is connected by an edge. Each edge is labeled with the evaluation result of the attribute test. When an input value is given, the edge with the same label as the result of evaluating the attribute test is selected. Hereafter, the evaluation result of the attribute test is called the “test value.” If the child node connected to the selected edge is a decision node, the same procedure is performed for that decision node. If the child node is a leaf node, the
value associated with that leaf node becomes the name of the class to which the input value belongs. As for the decision trees making up the DTEM, numeric values are used as class names, and those numeric values are used to calculate the output value of the DTEM. Hereafter, this value is called “the decision value.”

Arbitrary decision tree \( t \) constituting \( M \) can be expressed in the form \( (N_d, N_l, n_1, E, \text{attr}, tv, dv) \), where \( N_d \) represents a set of decision nodes, \( N_l \) represents a set of leaf nodes, and \( n_1 \) represents a root node included in \( N_d \). \( E \) is a set of edges, each of which edge is represented by a pair consisting of a connection-source node and a connection-destination node. That is, it can be defined as \( E \subseteq N_d \times (N_l \cup (N_d \setminus \{n_1\})) \).

Here, \( \text{attr} : N_d \rightarrow \mathbb{A} \) is a function that associates an attribute test with a decision node. \( A \) represents a set of arbitrary expressions for input value \( x \). Similarly, \( tv : E \rightarrow V \) is a function that associates a test value with an edge, where \( V = \{\text{True}, \text{False}\} \). Here, \( dv : N_l \rightarrow \mathbb{R} \) is a function that associates a decision value with a leaf node. The decision tree is acyclic. This is defined, by using transitive closure \( E^+ \) of \( E \), as \( \forall n \in N_d \cup N_l \cdot (n, n) \notin E^+ \).

It is assumed that the decision tree that constitutes the DTEM, which is the subject of this paper, has more than one decision nodes, and branches from a decision node to at least two child nodes. Therefore, if the number of nodes included in \( N_d \) is expressed as \( \text{card}(N_d) \), \( \text{card}(N_d) \geq 1 \) holds. In a similar manner, \( \text{card}(N_l) \geq 2 \) is established. As for the number of edges, \( \text{card}(E) \geq 2 \) holds.

B. Decision Tree Ensemble Model(DTEM)

Arbitrary \( M \) can be expressed as \( (T, \text{ensm}) \). \( T \) represents a set of decision trees constituting \( M \). If the number of decision trees included in \( T \) is taken as \( \text{card}(T) \), the decision trees included in \( T \) can be expressed as \( t_1, t_2, \ldots, t_{\text{card}(T)} \). The decision values of trees \( t_1, t_2, \ldots, t_{\text{card}(T)} \) are represented by \( y_1, y_2, \ldots, y_{\text{card}(T)} \), respectively.

Moreover, \( \text{ensm} \) represents a function that takes \( y_1, \ldots, y_{\text{card}(T)} \) as arguments and returns \( y \). The specification of \( \text{ensm} \) depends on the implementation of DTEM. For example, in the case of random forests, the average of the decision values is \( y \). Alternatively, in the case of a gradient-boosting decision tree, the sum of the decision values is \( y \). In this paper, the calculations depending on the implementation of the DTEM are generalized with \( \text{ensm} \).

III. OUTLINE OF PROPOSED METHOD

The proposed method is outlined in Algorithm 1. As for the proposed method, \( M \) is taken as the DTEM to be verified. A property of \( M \) is denoted by \( \varphi \), and \( X \) represents a domain of \( M \). Parameters \( r_a, r_b \), and \( r_c \) are described later. Algorithm 1 outputs the value of variable \( vranges \), which is a set of violation ranges for \( \varphi \). Function \( \text{input is within} \) on line 2 returns formula \( \rho \), meaning that “input value \( x \) of \( M \) is included in domain \( X \).” Procedure \( \text{formal verification} \) on line 4 verifies whether an input value that violates \( \varphi \) and satisfies constraint \( \rho \). If such an input value is detected, that value is assigned to variable \( ce \). If no such input value is detected, “None” is assigned to \( ce \). Procedure \( \text{formal verification} \) is described in detail in Section IV. Procedure \( \text{range extraction} \) on line 6 extracts the violation range for \( \varphi \) from around \( ce \). The extracted range is assigned to variable \( vio \). Moreover, a range in which no input value violates \( \varphi \) is extracted by the process of extracting \( vio \); therefore, a set of such ranges is output as variable \( \text{novios} \). Hereafter, a range in which no violating input value exists is referred to as a “no-violation range.” Procedure \( \text{range extraction} \) is shown in detail in Section V.

The extracted violation range, \( vio \), is divided as follows. First, on line 7 \( vio \) is copied to variable \( core \). On line 8 \( \text{continue division} \) is a function that returns a decision of whether to execute procedure \( \text{range division} \) on line 11 Parameter \( r_b \) is used as a reference value for the decision. Although the details of \( \text{continue division} \) are not specified here, for example, it can be implemented so as to return “True” if the volume of core is equal to or more than \( r_b \) percent of the volume of \( X \). On line 11 procedure \( \text{range division} \) divides \( core \) into multiple pieces. This division is performed on the basis of a no-violation range, which is an element of \( \text{novios} \). Of the ranges obtained by division, the inner range including \( ce \) (which is the starting point of range extension) is taken as a new core, and the set of other outer ranges is taken as \( \text{surrds} \). The \( \text{range division} \) is explained in detail in Section VI.

If \( \text{novios} \) contains a number of no-violation ranges, it is

```
Algorithm 1 Violation Ranges Detection

Input: \( M, \varphi, X, r_a, r_b, r_c \)
Output: \( vranges \)

1: \( vranges \leftarrow \emptyset \)
2: \( \rho \leftarrow \text{input is within}(M, X) \)
3: \( \text{while True do} \)
4: \( ce \leftarrow \text{formal verification}(M, \varphi, \rho) \)
5: \( \text{if } ce \neq \text{None then} \)
6: \( \text{vio, novios} \leftarrow \text{range extraction}(ce, M, \varphi, \rho, r_a, X) \)
7: \( \text{core} \leftarrow \text{vio} \)
8: \( \text{while continue division}(\text{core}, X, r_b) \)
9: \( \text{iv} \leftarrow \text{novios}.\text{pop}() \)
10: \( \text{core, surrds} \leftarrow \text{range division}(\text{core, iv, } \rho, r_c, M, \varphi) \)
11: \( \text{vranges} \leftarrow \text{vranges} \cup \text{surrds} \)
12: \( \text{end while} \)
13: \( \text{vranges} \leftarrow \text{vranges} \cup \text{core} \)
14: \( \rho \leftarrow \rho \land \text{input is outside}(M, \text{vio}) \)
15: \( \text{else} \)
16: \( \text{break} \)
17: \( \text{end if} \)
18: \( \text{end while} \)
19: \( \text{return } vranges \)
```
used for the division from the range pushed to $novios$ last, which is the outermost no-violation range in terms of violating input value $ce$. Therefore, the ranges stored in $surnds$ do not include no-violation ranges because they are outside the outermost no-violation range. Accordingly, the elements of $surds$ are added to $vranges$ without dividing them further (line 12). On the contrary, the new $core$ might include a no-violation range. Therefore, on line 8 it is evaluated whether $core$ will be divided further. If $core$ is not divided any more, it is added to $vranges$.

Then, on line 15 function $input_is_outside$ is used to create a constraint representing “Input value $x$ of $M$ is out of the range indicated by $vio$,” and that constraint is conjunctively appended to $rho$. After that, by re-executing formal verification on line 9 it is verified whether an input value violates $phi$ outside $vio$. If an input value that violates $phi$ is detected as a result of the verification, the violation range is extracted and divided by the same procedure. $vranges$ extracted by Algorithm 1 represents the range in which an input value violates $phi$. Therefore, by creating an input filter (shown in Fig. 1) based on $vranges$, it is possible to filter the violating input value.

IV. FORMAL VERIFICATION OF DECISION TREE ENSEMBLE MODEL

The procedure of formal verification in Algorithm 1 is shown in detail hereafter. For any decision tree $t_i \in T$ that constitutes DTEM $M$, a set of all paths extractable from $t_i$ is denoted by $P_i$. As stated in Section II-A, since a decision tree is assumed to be acyclic, the elements of $P_i$ are paths of finite length. Arbitrary path $p \in P_i$ can be represented by a sequence of nodes, $n_{p,1}^i, n_{p,2}^i, \ldots, n_{p,d}^i, n_{p+1}^i$. Among these nodes, $n_{p,1}^i, \ldots, n_{p,d}^i$ represent decision nodes, and $n_{p,d+1}^i$ represents a leaf node. Since $t_i$ has one or more decision nodes, $1 \leq d$ holds. Here, the specification of an arbitrary path $p$ of $t_i$ is defined as $f_p^i$ by using functions $attr$, $tv$, and $dv$ (defined in Section II-A) as follows:

\[
 f_p^i \stackrel{\text{def}}{=} \bigwedge_{j=1}^{d} \text{attr}(n_j^i)(x) = tv((n_j^i, n_{j+1}^i)) \rightarrow (y_i = dv(n_{d+1}^i)) \tag{1}
\]

The argument $(n_j^i, n_{j+1}^i)$ of function $tv$ represents an edge between nodes $n_j^i$ and $n_{j+1}^i$. By using $f_p^i$, the specification $F_i$ of $t_i$ is expressed as follows:

\[
 F_i \stackrel{\text{def}}{=} \bigwedge_{p \in P_i} f_p^i \tag{2}
\]

As explained in Section II-B, the output value $y$ of $M$ is calculated by $ensem$ from decision values $y_1, \ldots, y_i, \ldots, y_{\text{card}(T)}$. Therefore, the specification of $M$ is expressed by $F_M$ by using $ensem$ and $f_p^i$ as follows:

\[
 F_M \stackrel{\text{def}}{=} \left( \bigwedge_{i=1}^{\text{card}(T)} F_i \right) \land y = ensem(y_1, \ldots, y_i, \ldots, y_{\text{card}(T)}) \tag{3}
\]

Furthermore, $F$ is defined by using $phi$ (i.e., a property to be satisfied by $M$) and constraint $rho$ (which indicates that $x$ is included in domain $X$) as follows:

\[
 F \stackrel{\text{def}}{=} F_M \land \neg \phi \land \rho \tag{4}
\]

$F$ is input into the SMT solver to determine its satisfiability. If $F$ is unsatisfiable, it is guaranteed that no input value satisfies $phi$ in the range constrained by $rho$. If $x$ and $y$ satisfying $F$ are detected, the detected $x$ is returned as the violating input value, $ce$. It is thus possible to verify whether $M$ satisfies $phi$ by encoding $M$ to formula $F$ and solving the satisfiability problem by using the SMT solver.

V. VIOLATION RANGE EXTRACTION

A. Overview

As for range extraction, a violating input value is searched in the range around violating input value $ce$ as a starting point by using formal verification. If a violating input value is detected, the range is extracted as the violation range. Here, to promote intuitive understanding, it is assumed that variable $x$ representing the input value of $M$ is represented by a two-dimensional vector, namely, $[x[0], x[1]]$. The method for extracting the violation range by range extraction is outlined in Fig. 3. The violation range is extended by alternately increasing the values of $x[0]$ and $x[1]$. In step (1), the violating input value obtained on line 3 of Algorithm 1 is set as the initial value of the violation range. Next, in step (2), a search range is set from the initial value to upper direction of $x[0]$, and whether a violating input value exists within that
search range is verified by the method described in Section [IV]. If a violating input value is detected in the search range, as shown in step (3), the search range is imported into the violation range. In step (4), the search range is set to upper direction of \( x[1] \). From then on, the same procedure is repeated to extend the violation range. Although omitted in Fig. [3] in a similar manner as described above, the violation range is also extended to lower directions of \( x[0] \) and \( x[1] \).

As for range extraction, at the same time the violation range is extracted, no-violation ranges existing in that range are extracted. The method of extracting no-violation ranges is outlined in Fig. [4]. In step (1) of Fig. [4] the search range is set in the \( x[0] \) upper direction. Here, it is assumed that no violating input value exists in the search range. In that case, the no-violation range is created as shown in step (2). After the violation range is extended in the \( x[1] \) upper direction in step (3), the search range is reset in the \( x[0] \) direction in step (4). If a violating input value still does not exist in the search range, the no-violation range set in step (2) is extended by step (5). It is assumed that the violation range can be extended in steps (6), (7), and (8). In that case, the extension of the no-violation range is ended. Accordingly, when no violating input value exists in the search range, the search range is extracted as the no-violation range. As shown in step (9), a plurality of no-violation ranges may be created. One of the features of the proposed method is to extract this no-violation range. The extracted no-violation range is the utilized in range division described in Section [VI].

Regarding the proposed method, as a strategy for extending the violation range, alternately increasing the values of \( x[0] \) and \( x[1] \) is adopted. Other strategies, however, may be considered to extend the violation range. For example, a strategy by which the values of \( x[0] \) and \( x[1] \) are increased simultaneously can be considered. Strategies for extending the violation range are discussed in Section [VIII].

### B. Algorithm

The procedure of range extraction is shown in detail in Algorithm 2. Algorithm 2 returns \( \text{vio} \), which represents a violation range for \( \varphi \), and \( \text{novios} \), which represents a range in which no violating input value exists. The violation range can be defined by the lower-limit values and the upper-limit values.

**Algorithm 2 Algorithm of range_extraction**

**Input:** \( ce, M, \varphi, \rho, ra, X \)

**Output:** \( \text{vio}, \text{novios} \)

1: \( \text{vio}[\text{lower}] \leftarrow \text{ce} \)
2: \( \text{vio}[\text{upper}] \leftarrow \text{ce} \)
3: \( \text{novios} \leftarrow \emptyset \)
4: \( \text{tmp_vio}[\text{lower}] \leftarrow [\text{None}] \ast s \)
5: \( \text{tmp_vio}[\text{upper}] \leftarrow [\text{None}] \ast s \)
6: \( \text{cont_flag} \leftarrow \text{True} \)
7: **while** \( \text{cont_flag} = \text{True} \) **do**
8: \( \text{cont_flag} \leftarrow \text{False} \)
9: **for** each \( k \) in \( \{0, ..., s - 1\} \) **do**
10: \( \text{rtv_upper} \leftarrow \text{EXPAND}(\text{upper}, k, \text{vio}, \text{tmp_vio}, \rho, M, \varphi, X) \)
11: \( \text{rtv_lower} \leftarrow \text{EXPAND}(\text{lower}, k, \text{vio}, \text{tmp_vio}, \rho, M, \varphi, X) \)
12: **if** \( \text{rtv_upper} = \text{True} \vee \text{rtv_lower} = \text{True} \) **then**
13: \( \text{cont_flag} \leftarrow \text{True} \)
14: **end if**
15: **end for**
16: **end while**
17: **return** \( \text{vio}, \text{novios} \)

18: **procedure** \( \text{EXPAND}(\text{dir}, k, \text{vio}, \text{tmp_vio}, \rho, M, \varphi, X) \)
19: \( \text{sr} \leftarrow \text{vio} \)
20: **if** \( \text{dir} = \text{upper} \) **then**
21: \( \text{sr}[\text{dir}][k] \leftarrow \text{vio}[\text{dir}][k] + \text{mgn}(ra)[k] \)
22: **else** \( \text{dir} = \text{lower} \)
23: \( \text{sr}[\text{dir}][k] \leftarrow \text{vio}[\text{dir}][k] - \text{mgn}(ra)[k] \)
24: **end if**
25: \( \text{sr}[\text{opposite}(\text{dir})][k] \leftarrow \text{vio}[\text{dir}][k] \)
26: \( \text{sr} \leftarrow \text{intersection}(X, \text{sr}) \)
27: \( \rho' \leftarrow \rho \wedge \text{input is within}(M, \text{sr}) \)
28: \( ce' \leftarrow \text{formal_verification}(M, \varphi, \rho') \)
29: **if** \( ce' \neq \text{None} \) **then**
30: \( \text{vio}[\text{dir}][k] \leftarrow \text{sr}[\text{dir}][k] \)
31: **if** \( \text{tmp_vio}[\text{dir}][k] \neq \text{None} \) **then**
32: \( \text{novios}.\text{push}(\text{tmp_vio}[\text{dir}][k]) \)
33: \( \text{tmp_vio}[\text{dir}][k] \leftarrow \text{None} \)
34: **end if**
35: **return** \( \text{True} \)
36: **else**
37: \( \text{tmp_vio}[\text{dir}][k] \leftarrow \text{sr} \)
38: **return** \( \text{False} \)
39: **end if**
40: **end procedure**
of variables $x[0], ..., x[s-1]$. Therefore, it is supposed that violation range $vio$ is composed of $vio[lower]$ and $vio[upper]$, where $lower$ and $upper$ represent 0 and 1 respectively. Here, the lower-limit value of variables $x[k]$ ($0 \leq k \leq s-1$) is represented by $vio[lower][k]$. Similarly, the upper-limit value is represented by $vio[upper][k]$.

First, on lines 1 and 2, $vio$ is initialized with $ce$. On line 7, whether $vio$ has been extended in the previous loop is checked, and if $vio$ has been extended, a further extension is tried. On line 9, the variable for which the range is to be expanded, $x[k]$, is selected. For the selected $x[k]$ here, line 10 attempts to expand the upper limit of the variable, and line 11 tries to expand the lower limit of the variable. If neither the upper limit nor the lower limit can be expanded for all variables $x[0], ..., x[s-1]$, cont_flag = False is returned, and the procedure ends.

The upper and lower limits are extended on lines 10 and 11 by EXPAND. The procedure of EXPAND is described below with the case that the upper limit is expanded taken as an example. The parameter $dir$ of EXPAND represents the direction of expansion. That is, in this example, $upper$ is passed as an argument. On line 19 variable $sr$ is initialized with $vio$, where $sr$ represents the search range. On lines 21 and 25 $sr$ for variable $x[k]$ is updated. $opposite$ is a function that returns the direction opposite to the extension direction indicated by $dir$. $mgn$ is a function that accepts parameter $ra$ as an argument and returns a vector of the same length as $x$. Vector $mgn(ra)$ is used to set the search range of $x[k]$ to the range $vio[upper][k] < x[k] \leq (vio[upper][k] + mgn(ra)[k])$. Because the search range of variable $x[k'] (k' \neq k)$ is not updated, it is given as $vio[lower][k'] \leq x[k'] \leq vio[upper][k']$.

As a result, $sr$ is created on the upper bound of $vio$ in the $x[k]$ direction.

When $sr$ is defined, the upper-limit or lower-limit value of $sr$ is sometimes not included in the search range, that is, the search range is defined by using “$\geq$” or “$\leq$”. Accordingly, it is necessary to hold that information about with-equal or without-equal. In this paper, to simplify explanation, it is omitted how to keep that information in $sr$. The function $intersection$ on line 26 redefines the common range of $X$ and $sr$ as a new $sr$. As the violation range is extended, $sr$ is eventually created outside $X$, and then their common ranges become empty. In this way, the termination of the procedure is assured.

On line 27 $\rho'$ is created by conjunctively appending the constraint “the input value for $M$ is included in $sr'$” to $\rho$. Line 28 verifies whether $M$ satisfies $\varphi$ under constraint $\rho'$. As a result of verification, if the violating input value is detected within $sr$, the upper limit of the violation range, $vio[upper]$, is extended to $sr[upper]$ (line 30).

Lines 31, 33, and 35 create an element of $novios$. If no violating input value is detected in the search range as a result of the verification on line 28, the search range is saved in $tmp_nv[upper][k]$ (line 37). $tmp_nv[upper][k]$ stores a tentative no-violation range in the $x[k]$ upper direction. The range stored in $tmp_nv[upper][k]$ will be extended as long as a violating input value is not detected in the $x[k]$ upper direction (line 37). If a violating input value is detected in that direction, the latest range stored in $tmp_nv[upper][k]$ is finalized as a no-violation range and added to $novios$ (line 32).

VI. VIOLATION RANGE DIVISION

A. Overview

As for $range\_division$, violation range $core$ extracted by $range\_extraction$ is divided. The division is based on a no-violation range, which is an element of $novios$. Among the divided ranges, the range including violating input value $ce$ (which is the starting point of the range extension) is set as a new $core$, and the other ranges are called $surr$s. If $sur$ includes a violating input value, it is added to $surrds$. Since it is possible to repeatedly divide $core$, the number of ranges in $surrds$ may be increased with each division. For example, as for the violation range shown in step (9) in Fig. 4, the method of $range\_division$ is outlined in Fig. 5. In Fig. 5 first, the violation range is divided at the upper-limit value of the no-violation range on the $x[1]$ axis. Next, the range is divided at the upper-limit value of the no-violation range on the $x[0]$ axis. Finally, the range is further divided by using the lower-limit value of the no-violation range on the $x[0]$ axis. As a result this division, the violation range is divided into three ranges: $sur(i)$, $sur(ii)$, and $core$.

It is clear that there exists a violating input value in $core$ because $core$ includes violating input value $ce$ at least. $sur(i)$ also includes another violating input values in this case since the violation range was extended as shown in step (8) in Fig. 4. However, $sur(ii)$ may not include a violating input value. Whether $sur(ii)$ include a violating input value can be checked by using $formal\_verification$. If $sur(ii)$ does not include a violating input value, it is removed from the violation range.

This method aims to narrow the violation range by dividing it and verifying whether outer ranges $surs$ created by the division include violating input values. This is based on the assumption that outer ranges of the no-violation range are also likely not to include a violating input value. Outer ranges are “adjacent to” the no-violation range and also “beyond” the no-violation range from the starting point of the range extension. Accordingly, input values in outer ranges are assumed to be “similar to” the input values in the no-violation range and also “more different” from violating input values around the starting point than the input values in the no-violation range. Therefore, we consider that outer ranges created by
the division on the basis of the no-violation range are likely to include no violating input value.

When dividing the violation range on the basis of the no-violation range, the order of the division can be changed. For example, the dividing order shown in Fig. 6 can be considered. In this case, the division is performed in the following order: at the upper-limit value of the non-violating range on the $x[0]$ axis, at the upper-limit value on the $x[1]$ axis, and at the lower-limit value on the $x[0]$ axis.

![Fig. 6. Another order of dividing based on no-violation range](image)

It is said that the dividing order that makes the violation range narrower is more effective. However, the extent to which the violation range can be narrowed by a certain dividing order depends on the DTEM and property to be verified. Accordingly, as for the proposed method, multiple divisions are tried at random, and the order that makes the violation range the narrowest is adopted.

The violation range is divided by the surface of the non-violation range (corresponding to the dotted lines in Fig. 5 and Fig. 6). Since the no-violation range is an $s$-dimensional hyperrectangle, the number of dividing planes is $2s$. Therefore, the number of ways to divide the violation range is $(2s)!$. As for the proposed method, dividing orders, whose number equals parameter $r_c$, are randomly selected and the results of the divisions are compared according to the sum of the hypervolumes of the ranges in surrds. The dividing order that minimizes the sum of the hypervolumes is adopted.

B. Algorithm

The procedure of range_division is shown in details in Algorithm 3. In Algorithm 3, violation range core is divided on the basis of no-violation range iv. Then, as a result of division, a new core and surrds are returned. On line 1 calc_planes extracts the $(s-1)$-dimensional hyperplanes constituting hyperrectangle iv, and creates the set planes. The number of elements in planes is $2s$. On line 2 permutate creates as many arbitrary dividing orders using elements of planes as $r_c$, and the set of the created dividing orders is assigned to orders. Line 4 picks out arbitrary element od of orders. od is an array (with length of $2s$) having the hyperplanes constituting iv as elements. On line 7, the top element of od is assigned to plane. By slice_core on line 8 core is divided with plane as the dividing plane. Of the two ranges created by the division, the range that includes ce is taken as the new core, and the other is taken as sur. formal_verification on line 10 verifies whether the violating input value is included in the range of that sur. If a violating input value exists within the range of sur, sur is added to surrds (line 12). The division with plane is repeated for each element of od. On line 15 calc_volume_sum is used to calculate the sum of the hypervolumes of the violation range represented by the elements of core and surrds. The procedures from lines 5 to 16 are repeated according to the number of elements of orders, that is, $r_c$ times. In this manner, for each dividing order randomly created on line 2 core and surrds, which are division results, and volume, which is the sum of the hypervolumes, are obtained. On line 18 the result of division that gives the smallest volume is extracted by function min_volume, and is used as the final result value.

In this way, the violation range is divided by range_division on the basis of the no-violation range created by range_extraction. It should be noted that range_division is not executed if no-violation range is created by range_extraction.

VII. CASE STUDY

A. Setup

As a subject of the case study, DTEM $M$ is created by using a dataset of house prices \(^1\) as a training data set. XGBoost was adopted to implement $M$. M receives a 7-dimensional vector, $x = [x[0], ..., x[6]]$, as an input value, and returns a house price as output value $y$, where $x[0], ..., x[6]$ are, respectively, grade of house, condition of house, number of bedrooms, size of living room, size of parking space, size of ground floor, and size of basement. The number of decision

\(^1\)https://www.kaggle.com/harlfoxem/housesalesprediction

```python
Algorithm 3 Algorithm of range_division
Input: core, iv, $\rho$, $r_c$, $M$, $\varphi$
Output: core, surrds

1: planes ← calc_planes(iv)
2: orders ← permutate(plane, $r_c$)
3: results ← ∅
4: for each od in orders do
5:   core ← core
6:   while od ≠ ∅ do
7:       plane ← od.pop()
8:       core, sur ← slice_core(core, plane)
9:       $\rho' ← \rho ∧ \text{input_is_within}(M, sur)$
10:      $ce' ← \text{formal_verification}(M, \varphi, \rho')$
11:      if $ce' \neq \text{None}$ then
12:         surrds ← surrds ∪ sur
13:      end if
14:   end while
15:   volume ← calc_volume_sum(core, surrds)
16:   results ← results ∪ \{((core, surrds), volume)\}
17: end for
18: (core, surrds) ← min_volume(results)
19: return core, surrds
```
trees constituting $M$ is 100, and the maximum depth of each decision tree is 3. As properties to be verified, $\varphi_1$, $\varphi_2$, and $\varphi_3$ are defined as follows:

\[
\varphi_1 \overset{\text{def}}{=} x[0] \geq 7000 \Rightarrow y \geq 500000 \quad (5)
\]
\[
\varphi_2 \overset{\text{def}}{=} y > 50000 \quad (6)
\]
\[
\varphi_3 \overset{\text{def}}{=} y < 10000000 \quad (7)
\]

Here, $x[0]$ in $\varphi_1$ represents size of living room. The larger the living room, the higher the price. Therefore, it is defined as $\varphi_1$ that if the size of the living room is 7000 or more, the price is 500,000 or more. Moreover, the price output by $\varphi_2$ is taken as the property that the price is less than 50,000. Similarly, $\varphi_2$ is defined to express the property that the price is less than 10,000,000.

Next, domain $X$ of $x$ is defined. In this case study, $X$ is defined as the basis of the maximum and minimum values of the training dataset. For each variable $x[k]$ ($0 \leq k \leq 6$), the maximum value included in the training dataset is represented as $\max_k$, and the minimum value is similarly represented as $\min_k$. These values are used to define $X$ as follows:

\[
X \overset{\text{def}}{=} \{x[0], \ldots, x[k], \ldots, x[s-1]\} \\
\text{where} \quad \forall k \cdot \min_k \leq x[k] \leq \max_k \} \quad (8)
\]

Function $\text{mgn}$ (which determines the search range in range\_extraction) is implemented with parameter $r_a$ as an argument as follows:

\[
\text{mgn}(r_a) \overset{\text{def}}{=} [m[0], \ldots, m[k], \ldots, m[s-1]], \text{such that} \quad \\
\forall k \cdot m[k] = \max_k - \min_k / r_a \quad (8a)
\]
\[
\forall k \cdot m[k] = \text{ceil}((\max_k - \min_k) / r_a) \quad (x[k] \in \mathbb{R})
\]
\[
\forall k \cdot m[k] = \text{ceil}((\max_k - \min_k) / r_a) \quad (x[k] \in \mathbb{Z})
\]

(9)

Here, $\text{mgn}(r_a)[k]$ is created on the basis of the width of the domain $X$ (i.e., the difference between $\max_k$ and $\min_k$), where function $\text{ceil}$ rounds up a real number to an integer. In this case study, $r_a = 100$ is supposed. $\text{continue\_division}$ shown in Algorithm 1 is implemented so as to return “True” if the hypervolume of $\text{core}$ is $r_b$ percent or more of the hypervolume of $X$. In this case study, $r_b = 10$ is also supposed. Furthermore, parameter $r_c$ for determining the number of elements of $\text{orders}$ in range\_division is taken as $r_c = 10$.

The proposed method was implemented in Python[2]. The Z3 Theorem Prover[36] was used as a SMT solver in formal\_verification. Moreover, this case study was performed on a Windows 10® PC equipped with two Intel® Core™ i7-8700 3.2-GHz processor with 6 cores, 16-GB memory.

### B. Results and Evaluation

The results of applying the proposed method are listed in Table 1. The violation range detected by the proposed method is shown in column (b) for each property shown in column (a). In column (b), the violation range before division by range\_division and the violation range after division are shown. In column (c), the hypervolume of each violation range is shown. Column (d) shows a value obtained by dividing the sum of hypervolumes of the violation range after division by the sum of hypervolumes of the violation range before division. This value is used to evaluate the effect of narrowing the violation range by range\_division. Column (e) shows the time elapsed to execute the proposed method. Column (f) shows the time taken to execute the SMT solver as part of the method. Column (g) shows the number of times of the SMT solver is executed (i.e., number of calls), and column (h) shows the result of dividing (f) by (g), that is, the average execution time per run of the SMT solver. As a result of executing formal\_verification for $\varphi_1$, a violating input value was detected. Then, starting from the detected violating input value, range\_extraction is executed, and the violation range shown in #1-1 is extracted. Moreover, by dividing this violation range by range\_division, the violation ranges shown in #1-1-1 to #1-1-5 were obtained. This division reduced the hypervolume of the violation range to about 74 percent of the hypervolume before the division.

Similarly, with respect to $\varphi_2$, the violation ranges shown in #2-1 and #2-2 were extracted by range\_extraction. In this case, the division by range\_division was not performed because the hypervolume of each violation range was less than $r_b = 10$% of the hypervolume of $X$. However, it is confirmed that when $r_b$ is 0.1, range\_division reduces the hypervolume of the violation range to about 33% of the violation range before the division. As for $\varphi_3$, no violating input value was detected. That is, it was confirmed that $M$ satisfies $\varphi_3$.

The above results demonstrate that the DTEM can be formally verified by the method described as formal\_verification, that is, encoding the DTEM in a formula and solving it with an SMT solver. Moreover, it was confirmed that the violation range can be extracted by range\_extraction. It was also confirmed that using range\_division makes it possible to narrow the violation range. It can be therefore be concluded that the feasibility of the proposed method was demonstrated.

Next, we evaluate the scalability of the proposed method. If total execution time and execution time of the SMT solver are focused on, it becomes clear that the time taken to execute the SMT solver occupies most of the total execution time. In other words, the execution time of the proposed method is considered to be largely dependent on the execution time of the SMT solver. As a factor that affects the execution time of the SMT solver, for example, the number and depth of the decision trees constituting $M$ can be considered. However, various heuristics are implemented and black-boxed in the SMT solver, so it is difficult to predict how much these factors affect the execution time of the SMT solver. Accordingly, in this study, we change the value of the factor considered to affect the execution time of the SMT solver and measure the execution time. The following experiment uses the same
settings as described in Section VII-A unless otherwise stated. Moreover, a practically acceptable execution time is assumed as 24 hours, and the execution is aborted if it exceeds 24 hours.

1) (1) Number of decision trees: The number of decision trees constituting $M$ is represented by $n_{\text{est}}$. In proportion to the increase of $n_{\text{est}}$, the number of paths of $M$ increases. The length of formula $F_M$ increases in proportion to the number of paths of $M$. That is, with the increase of $n_{\text{est}}$, the length of the formula $F_M$ increases by $O(n_{\text{est}})$. Therefore, it is considered that the execution time of the SMT solver tends to increase with the increase of $n_{\text{est}}$. The execution times of the proposed method when the value of $n_{\text{est}}$ was changed are listed in Table II. The results in Table II show that the execution time of the SMT solver tends to increase as $n_{\text{est}}$ increases. As for the verification of $\varphi_2$, timeout occurs when $n_{\text{est}} \geq 190$. If the number of executions of the SMT solver is focused on, it turns out that the SMT solver was executed frequently in the case of $\varphi_2$. In such a case, changing parameters $r_a$, $r_b$, and $r_c$ may reduce the number of executions of the SMT solver and, thereby, shorten the execution time.

Parameter $r_a$ determines the size of the search range in range extraction. In the implementation shown in Equation 9, the smaller the value of $r_a$ is, the larger the search range is. Therefore, the number of times to call $formal\_verification$ can be reduced if the value of $r_a$ is changed smaller. Note that, in that case, it is highly likely that the extracted violation range is wider than before decreasing $r_a$. That is, the extracted violation range includes more input values that do not violate the property. $r_b$ is referred to as a reference value for determining whether or not to execute range division. In the implementation shown in Section VII-A the larger the value of $r_b$ is, the lower the possibility of dividing the violation range is. Thus, the number of times to call $formal\_verification$ is reduced if the value of $r_b$ is changed larger. In that case, however, the possibility of narrowing the violation range is reduced. $r_c$ determines how many dividing orders are tried in range division. Therefore, if the value of $r_c$ is changed smaller, the number of times to call $formal\_verification$ decreases. We changed the values of these parameters to $r_a = 20$, $r_b = 30$, and $r_c = 5$ and then retried the verification of $\varphi_2$. Execution times before and after changing the parameters are listed in Table III. As shown in Table III by changing the values of $r_a$, $r_b$, and $r_c$, even in the case of $190 \leq n_{\text{est}} \leq 220$, execution of the SMT solver could be completed without incurring the timeout. This demonstrates that when execution cannot be completed within a practical time due to the number of executions of the SMT solver, the execution time can be shortened by adjusting the values of these parameters. It should, however, be noted that when these parameters are adjusted to reduce the execution time, the extracted violation range may become wider. In fact, from Table III it turns out that the total volume of the violation range is increased by changing the value of parameters. On the contrary, it may be possible to narrow the violation range more strictly by changing these parameters if it is acceptable that the execution time become longer. That is, by changing these parameters, it is possible to adjust the balance between the fineness of the result and the execution time.

2) (2) Depth of decision trees: The maximum depth of a decision tree is taken as $max_d$. When the depth of a decision tree increases by 1, the number of paths extracted from one decision tree is doubled. (Moreover, the length of each path also increases in proportion to $max_d$.) Therefore, the length of formula $F_M$ is of $O(2^{max_d})$. Accordingly, if $max_d$ increases, it is considered that the execution time of the SMT solver tends to increase. The execution times of the

| (a) Property | (b) Violation range | (c) $hypervolume$ sum before division | (d) Hypervolume sum after division / $hypervolume$ sum before division | (e) Total run time (s) | (f) Number of executions of SMT solver | (g) Average execution time per run of SMT solver (s) |
|---|---|---|---|---|---|---|
| Before division | 6977 $\leq x_0 \leq 13540$ | $0 \leq x_1 \leq 33$ | 320 $\leq x_2 \leq 1651359$ | $1 \leq x_3 \leq 4$ | 220 $\leq x_4 \leq 9410$ | 0 $\leq x_5 \leq 4620$ | 3.78E+20 |
| After division | 6977 $\leq x_0 \leq 13540$ | $0 \leq x_1 \leq 33$ | 320 $\leq x_2 \leq 1651359$ | $1 \leq x_3 \leq 4$ | 220 $\leq x_4 \leq 9410$ | 0 $\leq x_5 \leq 4620$ | 3.95E+19 |
| Before division | 7110 $\leq x_0 \leq 13540$ | $0 \leq x_1 \leq 33$ | 320 $\leq x_2 \leq 1651359$ | $1 \leq x_3 \leq 4$ | 220 $\leq x_4 \leq 9410$ | 0 $\leq x_5 \leq 4620$ | 1.60E+18 |
| After division | 7110 $\leq x_0 \leq 13540$ | $0 \leq x_1 \leq 33$ | 320 $\leq x_2 \leq 1651359$ | $1 \leq x_3 \leq 4$ | 220 $\leq x_4 \leq 9410$ | 0 $\leq x_5 \leq 4620$ | 1.21E+20 |
| Before division | 290 $\leq x_0 \leq 2101$ | $0 \leq x_1 \leq 33$ | 320 $\leq x_2 \leq 1651359$ | $1 \leq x_3 \leq 4$ | 220 $\leq x_4 \leq 9410$ | 0 $\leq x_5 \leq 4620$ | 1.21E+20 |
| After division | 290 $\leq x_0 \leq 2101$ | $0 \leq x_1 \leq 33$ | 320 $\leq x_2 \leq 1651359$ | $1 \leq x_3 \leq 4$ | 220 $\leq x_4 \leq 9410$ | 0 $\leq x_5 \leq 4620$ | 1.21E+20 |

**Table I**

Violation ranges extracted by the proposed method
proposed method when the value of max_d was changed are listed in Table V. It is clear from the results in Table V that the average execution time of the SMT solver increases as max_d increases. In particular, as for the verification of \( \varphi_1 \) and \( \varphi_2 \), the execution time increases by 10 to 100 times each time max_d increases by one. When time-out is invalided and \( \varphi_1 \) is verified with max_d = 5, it takes about 70 hours to complete execution, and the average execution time of SMT solver is about 46 seconds. From these results, it can be said that max_d has a big influence on the execution time of the proposed method.

3) Dimension of input value: The loop in line 9 of range extraction in which formal verification is called is executed by the number of dimension s. In addition, as s increases, the number of variables appearing in \( F_M \) also increases, that is, \( F_M \) becomes more complicated. For the reasons above, it is considered that the execution time of the SMT solver tends to increase with increasing s. In the house-price dataset used in this study, we can use up to 18 attributes as elements of \( x \). Therefore, these attributes was used to change dimension s of x in the range of 2 ≤ s ≤ 18. The execution times of the proposed method when the value of s was changed are listed in Table V. It is clear from the results in Table V that execution time tends to increases as s increases. However, the execution time sometimes decreases though s increases, for example, in the verification of \( \varphi_1 \), s = 14. In other cases, execution time increases rapidly as when s = 13 in the verification of \( \varphi_1 \). Therefore, the correlation of s with the execution time is considered to be weaker than that of n_est and max_d with execution time.

From the results in Table III Table IV and Table V, it can be concluded that n_est, max_d, and s are factors that increase the execution time of the proposed method. In particular, n_est and max_d are important factors that determines the applicability of the proposed method. Specifically, the proposed method is practical if n_est is less than around 200 and max_d is less than 5 at least.

VIII. DISCUSSION

As for the proposed method, the search range is set around the violating input value detected first, and if a violating input value exists within the search range, the search range is defined as the violation range. Furthermore, by setting the search range around the violation range, the violation range is expanded in the same manner. After the violation range is extracted, it is confirmed whether another violating input value exists outside the violation range. If another violating input value exists, the violation range is extracted in the same way starting from that violating input value. In this manner, it is assured that all violating input values fall within any violation range. Therefore, by creating the input filter shown in Fig. I based on the violation ranges extracted by the proposed method, all violating input values leading to system failures can be filtered. Moreover, as for the proposed method, the violation range can be narrowed by dividing the extracted violation range. Through a case study, the feasibility of the division of the violation range was confirmed. The extent to which the violation range can be narrowed depends on the DTEM or the property to be verified; thus, it is difficult to estimate the effectiveness of the division in advance. However, dividing the violation range is still useful to narrow the violation range on a trial basis.
In the case study described in Section VII-B, it was shown that the number of decision trees constituting the DTEM, \( n_{est} \), maximum depth of the decision trees, \( \text{max}_d \), and dimension \( s \) of the input value are factors that increase the execution time of the proposed method. Among those factors, \( \text{max}_d \) has the greatest influence on the execution time. On the other hand, the correlation of \( s \) with execution time is considered to be relatively weak. Here, the length of the formula \( F_M \) is focused on. As mentioned above, \( F_M \) becomes longer by exponential order with increasing \( \text{max}_d \). On the contrary, the length of \( F_M \) does not change even if \( s \) increases. From these facts, it is inferred that the execution time of the proposed method strongly depends on the length of \( F_M \).

The purpose of the proposed method is creating the filtering condition for the input filter. In that case, the proposed method is required to complete execution within a practical time. However, even if execution of the proposed method is not completed within a practical time, the proposed method can be utilized by outputting the violation range extended up to that point when the execution was interrupted. The violation range extracted at the time execution is suspended does not include all violating input values. However, it is useful for realizing where violating input values exist around. For example, it is possible to use the input values included in the violation range as training data for additionally training the DTEM. Thus, the proposed method can be useful, even in the case of a large-scale DTEM whose execution is not completed within the practical time.

As described in Section I as a method for obtaining the condition to filter input value that violates a property, a method that outputs all the violating input values can be considered. However, for a certain violating input value, be a large number of similar violating input values might exist, and the values of some elements in them might be slightly changed. Accordingly, from the viewpoint of calculation time, it is not practical to detect all violating input values. Furthermore, as

| \( n_{est} \) | Before changing parameters (\( n_0 = 100, \ n_h = 10, \ \rho_0 = 10 \)) | After changing parameters (\( n_0 = 20, \ n_h = 30, \ \rho_0 = 5 \)) |
|---|---|---|
| Total run time (s) | 710.31 | 706.85 |
| SMT solver run time (s) | 1097 | 1097 |
| Number of executions of SMT solver | 0.64 | 2.09E+18 |
| Average execution time per run of SMT solver (s) | 0.49 | 5.82E+18 |
| Hypervolume sum (after division) | 477.02 | 475.84 |
| Total run time (s) | 1052.60 | 1060.70 |
| SMT solver run time (s) | 1281.94 | 1218.44 |
| Number of executions of SMT solver | 0.36 | 7.36E+19 |
| Average execution time per run of SMT solver (s) | 0.32 | 1.992E+19 |
| Hypervolume sum (after division) | 1901.15 | 750 |

### TABLE III

**Execution times before and after changing parameters**

| \( \text{max}_d \) | \( \varphi_1 \) | \( \varphi_2 \) | \( \varphi_3 \) |
|---|---|---|---|
| Total run time (s) | SMT solver run time (s) | Number of executions of SMT solver | Average execution time per run of SMT solver (s) |
| 2 | 5,77 | 5,44 | 1657 | 0.003 | 30.03 | 24.54 | 1641 | 0.01 | 0.05 | 0.22 | 1 | 0.22 |
| 3 | 1063.66 | 1061.14 | 822 | 1.29 | 710.31 | 706.85 | 1097 | 0.64 | 1.53 | 0.94 | 1 | 0.94 |
| \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) |
| 4 | 1698.45 | 1695.46 | 738 | 25.01 | 5429.52 | 5475.24 | 2350 | 52.69 | 4.91 | 3.26 | 1 | 3.26 |
| 5 | 877.77 | 875.44 | 1657 | 0.003 | 30.03 | 24.54 | 1641 | 0.01 | 0.05 | 0.22 | 1 | 0.22 |
| 6 | 1771 | 1768.67 | 1417 | 1 | 61.89 | 1 | 61.89 |
| 7 | 2145.7 | 2068.94 | 1 | 700.16 | 693.96 | 1 | 693.96 |
| 8 | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) |
| 9 | 2465.35 | 2463.96 | 1 | 4579.16 | 4544.48 | 1 | 4544.48 |
| 10 | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) |
| 11 | 7644.85 | 7601.93 | 1 | 5123.83 | 5192.93 | 1 | 5192.93 |
| 12 | 2906.68 | 2914.78 | 1 | 2914.78 | 2914.78 |
| 13 | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) |
| 14 | 5540.85 | 55320.07 | 1 | 55320.07 |
| 15 | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) | \( \text{timeout} \) |

### TABLE IV

**Execution times in case of changing \( \text{max}_d \)**
another approach, extracting the path condition \[\varphi_1\] of the decision trees constituting DTEM is considered hereafter.

First, in the same manner as the proposed method, the violating input value is detected by verification. If the detected violating input value is input into the DTEM, the execution path of each decision tree is uniquely determined. Here, the condition of the input value to execute these paths is called “path condition”. The path condition can be extracted by executing the DTEM with the violating input value or by analyzing the decision trees. If an input value satisfies the path condition and it is input into the DTEM, the same paths are executed as they are when the violating input value is input. Moreover, if the same paths are executed, the same output value is returned. Therefore, for all input values satisfying the path condition, the DTEM returns the same output value. Here, by assigning the output value to variable \(y\) appearing in the property to be verified and negating the assigned property, a conditional expression for the input value is created. For example, let the output value be 499, 999, the conditional expression of property \(\varphi_1\) is \(\neg(x[0] \geq 7000 \Rightarrow 499999 \geq 500000)\). If an input value satisfies the conjunction of the path condition and the conditional expression of the property, it means that the corresponding output value is 499, 999, and at that time, the property is not satisfied. Hereafter, the conjunction is called the “violation condition.” By repeating the above procedure, other violation conditions can be extracted. Finally, the set of the extracted violation conditions can be used as the filtering condition of the input filter.

Since the violation condition extracted by this method filters only input values that violate the property, this method creates the filtering condition more precisely than the proposed method. However, from the viewpoint of calculation time, this method is considered to be less practical than the proposed method because it is likely to extract a large number of violation conditions. For example, in the case of a DTEM composed of 100 decision trees with two paths each, there are \(2^{100}\) combinations of paths, that is, about \(10^{29}\) combinations. The violation condition extracted in a single verification is only one of the \(10^{29}\) combinations, and it is highly likely that there will be a large number of similar violation conditions in which part of the path condition differs. In that case, since the number of verifications by the SMT solver is also enormous, this method is not practical.

In general, the fineness of the result and the execution time have a trade-off relationship. As for the strategy to extend the violation range, this trade-off relationship should be discussed. As for the proposed method, the violation range is extended(154,182),(276,197) in each direction of \(x[k]\) \((0 \leq k \leq s - 1)\) in order. Hence, violation ranges are extracted in the form of hyperrectangles. The extension strategy of the proposed method is therefore called a “hyperrectangle strategy.” Here, the mesh strategy described below can be considered as a strategy for extracting the violation range more precisely than the hyperrectangle strategy. An example of extending the violation range according to mesh strategy is shown in Fig. 7.

![Fig. 7. Example of extending violation range by using mesh strategy](image)

### TABLE V

| \(s\) | \(\varphi_1\) | \(\varphi_2\) | \(\varphi_3\) |
|-----|-----|-----|-----|
| Total run time (s) | SMT solver run time (s) | Total run time (s) | SMT solver run time (s) | Total run time (s) | SMT solver run time (s) | Number of executions of SMT solver | Average execution time per run of SMT solver (s) | Number of executions of SMT solver | Average execution time per run of SMT solver (s) | Number of executions of SMT solver | Average execution time per run of SMT solver (s) |
| 2   | 0.31 | 0.03 | 2   | 0.31 | 0.03 | 2   | 0.31 | 0.03 | 2   | 0.31 | 0.03 | 2   | 0.31 | 0.03 |
| 3   | 0.66 | 0.31 | 3   | 0.66 | 0.31 | 3   | 0.66 | 0.31 | 3   | 0.66 | 0.31 | 3   | 0.66 | 0.31 |
| 4   | 0.77 | 0.43 | 4   | 0.77 | 0.43 | 4   | 0.77 | 0.43 | 4   | 0.77 | 0.43 | 4   | 0.77 | 0.43 |
| 5   | 62.37 | 61.08 | 5   | 62.37 | 61.08 | 5   | 62.37 | 61.08 | 5   | 62.37 | 61.08 | 5   | 62.37 | 61.08 |
| 6   | 553.87 | 549.83 | 6   | 553.87 | 549.83 | 6   | 553.87 | 549.83 | 6   | 553.87 | 549.83 | 6   | 553.87 | 549.83 |
| 7   | 1053.66 | 1061.14 | 7   | 1053.66 | 1061.14 | 7   | 1053.66 | 1061.14 | 7   | 1053.66 | 1061.14 | 7   | 1053.66 | 1061.14 |

**TABLE V** Execution times in case of changing dimension \(s\)
no violating input value exists in any mesh adjacent to the violation range, the extension of the violation range is ended. When this strategy is adopted, the number of adjacent meshes increases as the violation range is extended. For example, suppose that the violation range is an $s$-dimensional hypercube and the number of meshes on one side of the violation range is $\alpha$. In this case, the number of adjacent meshes to the violation range is $2s \cdot \alpha^{s-1}$. The verification is performed for the number of adjacent meshes to extend the violation range by one round. Therefore, the amount of calculation to extend the violation range by one round increases as the violation range is extended. On the other hand, in the case of the hyperrectangle strategy, to extend the violation range by one round in the same situation, the verification is performed only $2s$ times. That is, it advantageous that the violation range can be extended with a fixed amount of calculation regardless of the size of the violation range.

Moreover, a strategy to increase the values of $x[1], \ldots, x[s-1]$ simultaneously can be considered. This strategy can extend the violation range in a shorter time than the hyperrectangle strategy. In this case, the violation range is extracted in the form of a hypercube, so this strategy is called a “hypercube strategy.” An example of extending the violation range on the basis of the hypercube strategy is shown in Fig. 8. When the hypercube strategy is adopted, the number of verifications for extending the violation range by one round is one. However, the violation range extracted by the hypercube strategy is coarser than that of other strategies. An example of the extracted violation range for each strategy is shown in Fig. 9. The hypercube strategy is highly likely to include a range in which no violating input value exists in the violation range. Therefore, if an input filter is created based on the violation range created by the hypercube strategy, many input values satisfying the property will be filtered.

Finally, in terms of execution time, mesh strategies are considered less practical than hyperrectangle strategies. The hypercube strategy is considered to be less practical in terms of the fineness of the results. Therefore, we adopted a hyper-rectangle strategy with a good balance between the fineness of the result and execution time.

As mentioned in Section VII-B changing the values of $r_n$, $r_f$, and $r_c$ makes it possible to adjust the balance between fineness of the result and execution time. These parameters should be changed by the user according to the DTEM and property to be verified, and also allowable execution time. As for finding an appropriate parameter value, the following procedure can be considered. First, the parameter values are set so that execution time is short, and it is confirmed by actually executing the procedure that the execution time can be shorter than the allowable time. After that, the parameters are gradually changed to improve the fineness of the result. This procedure improves the fineness of the results within the constraints of execution time.

Among the proposed methods, range extraction and range division are applicable to machine-learning models other than a DTEM. On the contrary, formal verification is a procedure specialized for a DTEM, so it must be replaced with a verification method suitable for the target machine-learning model. For example, in the case of a DNN, methods for encoding and verifying the DNN in a formula have been proposed. By replacing formal verification in the proposed method with such a method, it is possible to extract the violation range of the DNN.

**IX. Related Work**

As described in Section I, Bogdiukiewicz et al. proposed a method to develop a policing function for autonomous systems. The policing function checks output values of intelligent function such as a machine learning model at runtime. The policing function is useful to prevent failures as well as the input filter. However, the policing function works after the intelligent function executes. It means that the policing function detects and controls the possible failure later than the input filter. Therefore, the proposed method for creating the input filter is more useful in development of systems which require quick handling of failures.

In formal verification shown in Section V, a rule-form formula composed of conditions and conclusions is extracted from the DTEM. Extracting a specification from a program in the form of rules has been studied. In these studies, the control path of the program is extracted by static analysis or symbolic execution, and the rule is created on the basis of the branch condition that constitutes the control path. The method proposed in this paper extracts the decision tree path and creates a rule-form formula based on the attribute test associated with the decision node that composes the path. It can thus be said that it adopts the same approach.

As for the verification of the rule-form formula, methods have been established to verify properties such as consistency, redundancy, and completeness. Furthermore, a method for extracting “minimal unsatisfiable subsets,” which are useful for
causal analysis when the rule set does not satisfy consistency, has also been proposed [31] [32] [33]. Validation of rule programs used in a business rule management system has also been studied [35]. As in the case of the method proposed in this paper, the verification of the rule-form formula is translated to checking the satisfiability problem.

Moreover, as described in Section I, an approach of encoding a DNN model in a formula and verifying it by determining the satisfiability of the formula has been proposed in recent years [15] [16] [17] [18] [14] [19]. It can be said that the method for verifying the DTEM proposed in this paper also takes this approach. However, a study describing a specific method for verifying a DTEM has not been reported. One of the contributions of this paper is formally specifying a verification method for a DTEM and demonstrating its feasibility through a case study.

No similar studies on extracting violation ranges as described in Section V and dividing violation ranges as described in Section VII have been reported. One of the reasons for that situation is that the problems solved by these methods are unique to software developed by machine learning. In regard to conventional software, namely, algorithmic programs, when a counterexample is detected by verification, the fault that caused the counterexample is analyzed, and the fault is removed by correcting the algorithm. Verification and correction are then repeated until no counterexample is detected. On the other hand, in the case of a machine learning model, a possible method for handling such faults is retraining or additional training using the detected counterexample (and data similar to it). Although this approach may eliminate the fault, retraining and additional training may affect the entire model, and another new fault may be inserted. That is, “regression” occurs. Although this regression also occurs in the case of algorithmic programs, in that case, regression occurs due to a developer’s mistake; therefore, such mistakes must be carefully corrected to avoid regression. On the contrary, in the case of a machine-learning model, it is difficult for the developer to control retraining and additional training so that regression does not occur. In other words, it can be said that it is inherently difficult to create a complete machine-learning model that always returns the expected output value. Accordingly, when a machine-learning model is implemented in a system, the proposed method is used to implement the input filter. That is, the proposed method solves a specific problem that occurs with the use of machine-learning models.

X. Conclusion

When a DTEM is implemented in a system, the input filter can be effectively used to prevent system failures. As a means of creating the filtering condition for the input filter, a method for extracting the violation range of the DTEM, whose input values are multi-dimensional vectors whose elements are numeric variables, was proposed. The proposed method consists of procedures for formally verifying the DTEM, extracting the violation range, and narrowing the extracted violation range. The violation range extracted by the proposed method includes all violating input values. The proposed method is therefore useful to create the filtering condition. After showing the algorithms for these procedures, the results of a case study using a dataset on the house prices were presented. On the basis of the results of this case study, the feasibility of the proposed method is demonstrated. Through the case study, the scalability of the proposed method is also evaluated. The number of decision trees constituting the DTEM, the maximum depth of the decision trees, and the dimension of the input value are factors that increase the execution time of the proposed method. Specifically, it was concluded that the proposed method is practical if the number of decision trees is less than around 200 and the maximum depth of the decision trees is less than 5 at least.

Future issues include improving the scalability of the proposed method. Since the form of the formula $F_M$ created by the proposed method is constant, the method’s scalability may be improved by implementing heuristics specialized for that form in the SMT solver. Moreover, the procedure for finding appropriate values of parameters $r_a$, $r_b$, and $r_c$ shown in Section VII can be incorporated into the proposed method. Furthermore, by adding case studies using other datasets, the limitation of the proposed method can be evaluated in more detail.

References

[1] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems, pages 1097-1105, 2012.
[2] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. IEEE Signal Processing Magazine, 29(6):82-97, 2012.
[3] L. Breiman: Random forests. Machining Learning, 45(1), 5-32, (2001).
[4] J.H. Friedman: Greedy function approximation: a gradient boosting machine. Annals of statistics, pp.1189-1232 (2001).
[5] P. Li: Robust logitboost and adaptive base class (abc) logitboost. arXiv preprint arXiv:1203.3491 (2012).
[6] M. Richardson, E. Dominowska, and R. Ragno: Predicting clicks: estimating the click-through rate for new ads. In Proceedings of the 16th international conference on World Wide Web, pp.521-530. ACM (2007).
[7] C.J.C. Burges: From ranknet to lambdarank to lambdamart: An overview. Microsoft Research Technical Report MSR-TR-2010-82 (2010).
[8] R.A.V. Rossel, and T. Behrens: Using data mining to model and interpret soil diffuse reflectance spectra. Geoderma 2010, 158, pp.46-54 (2010).
[9] P.T. Sorenson, C. Small, M.C. Tappert, S.A. Quideau, B. Drozdowski, A. Underwood, and A. Janz: Monitoring organic carbon, total nitrogen, and pH for reclaimed soils using field reflectance spectroscopy. Can. J. Soil Sci. 2017, 97, 241-248 (2017).
[10] A.M. Prasad, L.R. Iverson, and A. Liu: Newer classification and regression tree techniques: Bagging and random forests for ecological prediction. Ecosystems 2006, 9, pp.181-199 (2006).
[11] H. Ishwaran: Variable importance in binary regression trees and forests. Electron. J. Stat. 2007, 1, pp.519-537 (2007).
[12] C. Bogd'kiewicz, M. Butler, T.S. Hoang, M. Paxton, J.H. Snook, X. Waldron, and T. Wilkinson: Formal Development of Policing Functions for Intelligent Systems. In 2017 IEEE 28th International Symposium on Software Reliability Engineering (2017).
[13] T. Chen, and C. Guestrin: Xgboost: A scalable tree boosting system. In Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.785-794. ACM, (2016).
[14] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu.: Safety Verification of Deep Neural Networks, Computer Aided Verification 2017, Lecture Notes in Computer Science, vol.10426, pp.3-29 (2017).
