IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 14, 2020, accepted September 14, 2020, date of publication September 29, 2020, date of current version October 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3027508

Anomalies Detection in Software by Conceptual Learning From Normal Executions

AHMAD QADEIB ALBAN, FAHAD ISLAM, QUTAIBAH M. MALLUHI, (Member, IEEE), AND ALI JAOUA
Department of Computer Science and Engineering, Qatar University, Doha, Qatar

Corresponding author: Ali Jaoua (jaoua@qu.edu.qa)

This work was supported by the Qatar National Research Fund (Member of the Qatar Foundation) under Grant NPRP X-063-1-014. Open Access funding was provided by the Qatar National Library.

ABSTRACT Could we detect anomalies during the run-time of a program by learning from the analysis of its previous traces for normally completed executions? In this paper we create a featured data set from program traces at run time, either during its regular life, or during its testing phase. This data set represents execution traces of relevant variables including inputs, outputs, intermediate variables, and invariant checks. During a learning mining step, we start from exhaustive random training input sets and map program traces to a minimal set of conceptual patterns. We employ formal concept analysis to do this in an incremental way, and without losing dependencies between data set features. This set of patterns becomes a reference for checking the normality of future program executions as it captures invariant functional dependencies between the variables that need to be preserved during execution. During the learning step, we consider enough input classes corresponding to the different patterns by using random input selection until reaching stability of the set of patterns (i.e. the set is almost no longer changing, and only negligible new patterns are not reducible to it). Experimental results show that the generated patterns are significant in representing normal program executions. They also enable the detection of different executable code contamination at early stages. The proposed method is general and modular. If applied systematically, it enhances software resilience against abnormal and unpredictable events.

INDEX TERMS Anomaly detection, functional dependencies, formal concept analysis (*FCA*), data reduction, pattern generation, functional dependencies preservation.

I. INTRODUCTION

Data science and machine learning methods offer new methods for extracting knowledge and reducing big data while preserving the underlying main concepts in a summarized form. One may define a program by its preliminary specification, and then give a formal verification for its correctness through a mathematical analysis. But this is not sufficient. A program-testing step is necessary as the implementation may not accurately represent the theoretical algorithm because of programming errors. Even if the program, representing a mathematically proven algorithm, passes the testing step successfully, there is no guarantee about its permanent correct execution. In fact, the program environment might contaminate its state, by either transient or permanent errors, or by malicious external actors, which may cause an abnormal or unpredictable behavior of the program. In the case of critical systems, these errors, even if they appear once, might provoke human or environmental disasters. Waiting until the final execution to judge the correct behavior of such a program is not sufficient as it cannot undo wrong actions or remove the risk of unpredictable events. For example, due to damaged electrical signals or radiations, a robot may lose its equilibrium; an airplane may have a sudden direction change, a pressure valve may dangerously close in an industrial plant, etc.

During the last two decades, several researchers have built anomaly detection methods based on feature definition of the system parameters from which they learn how to recognize inconsistent or unstable states [1], [2]. On one hand, most empirical machine learning methods are able to provide limited accuracy in these methods. But on the other hand, strict mathematical specification of these systems [3] and their control states are not easy to handle completely. Some authors even tried to repair errors in the code [4], [5]. In this paper, instead of evaluating the state of the program on the basis of its mathematical specification, we adopt an original machine

The associate editor coordinating the review of this manuscript and approving it for publication was Junchi Yan.

learning approach by mining the features of a program's execution traces to recognize future anomalies. First, we incrementally generate some knowledge K containing consistent reduced conceptual sets of patterns associated with several correct executions of the program. For each random input, the program execution generates a set of traces Tr. These traces Tr are then mapped to a formal context FC that we include into the global set of patterns in a reduced form (the knowledge K). Implicitly, FC represents the list of functional dependencies between all attributes of the traces generated by one program execution K [6]–[8]. In this research, for the learning process, we try to find the minimum set of random inputs after which the gathered knowledge K becomes stable enough to be used for discovering anomalies. The stability of K is measured by the number of new patterns added to it over its total size after each program execution. In this work, we propose a novel approach to detect run-time program anomalies at the earliest stages, by first learning from its normal completed executions. We also validate the proposed method on several known algorithms such as sorting, searching, and minimum spanning tree extraction from a graph (i.e. Kruskal's approach).

In the following section II, we present the state of the art about anomalies detection in a program and show to what extent it is similar to anomaly detection in data. In section III, we give some background on Formal Concept Analysis (FCA) as an important theoretical foundation for data analysis and for generating functional dependencies from formatted data [6], [7]. In section IV, we explain how we can represent functional dependencies in a formatted relational database with a formal context in reduced form. After considering case studies, in section V, we develop a methodology for anomaly detection in two phases. In the first phase, we collect the set of patterns K reflecting the normal behavior of a program. This is built as the union of all conceptual patterns generated automatically by random inputs. In the second phase, we use the set of patterns K as a reference to find anomalies in the program. In the same section V, we study the cases of sorting algorithms [9] and Kruskal algorithm for a minimal spanning tree in a graph [10]. These cases are used for evaluating our proposed approach. In section VI, we discuss the general validation of the proposed method. Section VII concludes the paper.

II. STATE OF THE ART ABOUT ANOMALIES DETECTION IN PROGRAMS AND DATA

In [11], the authors propose a solution for detecting anomalies in software during its execution, by controlling software invariant preservation. The method is interesting but suffers from the inconvenience that the number of invariants in a piece of software is high, and their manual extraction is very time consuming. Authors in [3] wrote: " ..., the number of program invariants for large scale software is usually large, making it unfeasible to monitor all invariants online. Thus, an effective selection of invariants is essential. Existing selection approaches are mainly based on the types of invariants..., only invariants related to object read/write at each method invocation are selected, while ...the models created from machine learning approaches are used to rank the relevance of the invariants to being fault-revealing based on their types". In [12], Jianguang Lou et al proposed a log file analysis using learning method from data invariant relations. Recently, other authors proposed generating traces in a moderate way and checking some invariant, thus avoiding excessive splits of the program with too many checkpoints. In [4], Xi Cheng (2016) proposed anomaly detection and fault localization using run-time state models. In the literature, we also found research about feature extraction which constitutes a major problem to solve for the learning process [13]. In general, we could not find significant theoretical foundation for an automatic online analysis of software traces. In this paper, we mapped the problem of program anomalies detection to mining trace data sets anomalies either during the program execution or afterwords (i.e. by using a log file).

In an instance of a database, an abnormal object (i.e. row) is considered an outlier if it does not belong to any significant cluster, or does not respect some integrity conditions. In the context of supervised learning on formatted data, an object that we cannot classify in a particular category is an outlier. A more accurate definition of anomalies is associated to the state of the instance of the database as a whole. When incrementally updated, a database instance which changes its set of functional dependencies after each additional row is semantically unstable. It reaches a steady state if the pattern behind the list of satisfied functional dependencies becomes stable. This might serve as a measure for future abnormal states. This definition of a steady state in itself is not obvious and may only be approximate or used in some particular assumptions. By analogy, if we consider the list of structured traces of a program as an instance of a database, we can use existing machine learning methods for detecting anomalies in databases for finding corresponding anomalies in the program. However, these methods are not accurate enough because they are only based on correlations or statistics but not on exact dependencies as can be observed in the following two sections.

III. FORMAL CONCEPT ANALYSIS BACKGROUND

In this paper, during the learning phase, we use formal concept analysis for generating patterns from traces in a reduced form (i.e. knowledge K). An FCA-based algorithm checks the validity of a trace during the anomaly detection step with respect to the knowledge K. During the last three decennials, formal concept analysis theory emerged as a mathematical background for concepts structure representation of data, useful for extracting knowledge and finding concepts through "formal concepts" [14], [15]. Since its inception, we found an increasing number of applications for machine learning, data analysis, implications, and associations' extractions. We use it to build features from texts, and to find some significant decreasing importance order of objects or attributes in a binary context. Recently, it has been applied for context

reduction without losing implications corresponding to the functional dependencies defined between the attributes of pixels in the image [6], [7].

FCA starts from the philosophical view that a mapping of a context to a structured space of formal concepts may approximate real concepts representing texts or general structured or unstructured data mathematically. From the relativity of objects defined by conceptual clusters, and the logical structure behind the hierarchy of concepts, we can extract implications. It is possible to map most datasets, with different ways to a formal context by defining the spaces of objects, attributes, and a binary relation between these two spaces. This *FCA* structure is suitable for categorizing objects and attributes in a logical way.

Definition 1 (Formal Context): A formal context is a triplet $\langle O, A, R\rangle$, where O is a set of objects, A is a set of attributes and R, a binary relation linking some objects belonging to O to some attributes in A: $R \subseteq O \times A$.

Example: Context "Divides", is composed of objects O as numbers between 1 and 12, represented by the objects o_1, o_2,…,o_{12}, and attributes A consisting of the numbers $1, 2, \ldots, 6$ represented by the attributes $Attr_1, Attr_2, \ldots, Attr_6$, respectively. Where o_i and $Attr_j$ represents integer i as an object and integer j as an attribute respectively.

Definition 2 (Galois Connection): Galois Connection is defined by two dual operators, γ, and λ:
$X \subseteq O$,
$\gamma(X) := \{\alpha \in A | \forall o \in X : (o, \alpha) \in R\}$,
$Y \subseteq A$,
$\lambda(Y) := \{o \in O | \forall \alpha \in Y : (o, \alpha) \in R\}$. $\gamma(X)$ is the set of all attributes shared by all objects in X while $\lambda(Y)$ is the set of all objects that share all the attributes in Y. The two operators γ and λ define the Galois connection between sets O and A.

Definition 3 (Formal Concept): A pair (X, Y) is a formal concept of the formal context (O, A, R) if and only if: $X \subseteq O, Y \subseteq A, \gamma(X) = Y$ and $\lambda(Y) = X$

Remark: The closure operators $\gamma o\lambda$, and $\lambda o\gamma$ are important to build formal concepts, by either completing the set of objects X, by calculating $\lambda o\gamma(X)$, or a set of attributes Y, by calculating $\gamma o\lambda(Y)$.

Illustration: For the context in Fig. 1, we may find here below all formal concepts:

C1 $= \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9, o_{10}, o_{11}, o_{12}\}$ x $\{Attr_1\}$

$C2 = \{o_5, o_{10}\}$ x $\{Attr_1, Attr_5\}$

$C3 = \{o_4, o_8, o_{12}\}$ x $\{Attr_1, Attr_2, Attr_4\}$

$C4 = \{o_3, o_6, o_{12}\}$ x $\{Attr_1, Attr_3\}$

$C5 = \{o_6, o_{12}\}$ x $\{Attr_1, Attr_2, Attr_3, Attr_6\}$

$C6 = \{o_{12}\}$ x $\{Attr_1, Attr_2, Attr_3, Attr_4, Attr_6\}$

$C7 = \{o_{10}\}$ x $\{Att_1, Attr_2, Attr_5\}$

$C8 = \{o_2, o_4, o_6, o_8, o_{10}, o_{12}\}$ x $\{Attr_1, Attr_2\}$

$C9 = \{\}$x$\{Attr_1, Attr_2, Attr_3, Attr_4, Attr_5, Attr_6\}$

The total number of formal concepts is 9. In Fig. 2, we represent a line diagram of the lattice of concepts. Each circle represents a formal concept, and if two concepts are in the two extremities of a line, the concept at an higher position in the line diagram is "greater" than the one drawn at a lower position in the same path. Here we define the order relation between comparable two formal concepts. Let (X, Y) be a formal concept as in Def. 3. We call X the extension (i.e.domain) of the formal concept, and set Y its intention (i.e. range). Formal concepts $\{(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)\}$ mined from the formal context (O, A, R) can be structured as a lattice of concepts. The sub-concept order relation naturally arranges the formal concepts of a given context into a lattice; the super-concept relationship is defined by $(X_1, Y_1) \leq (X_2, Y_2)$ if and only if $(X_2 \subseteq X_1)$ and $(Y_1 \subseteq Y_2)$.

"Divides"	$Attr_1$	$Attr_2$	$Attr_3$	$Attr_4$	$Attr_5$	$Attr_6$
o_1	X					
o_2	X	X				
o_3	X		X			
o_4	X	X		X		
o_5	X				X	
o_6	X	X	X			X
o_7	X					
o_8	X	X		X		
o_9	X		X			
o_{10}	X	X			X	
o_{11}	X					
o_{12}	X	X	X	X		X

FIGURE 1. Example of a formal context.

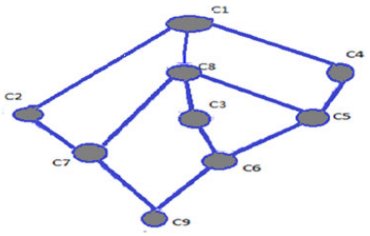


FIGURE 2. Lattice of concepts associated to Fig. 1.

Remark: Intentions of concepts in the same line diagram path increase, while their extensions shrink with respect to the inclusion order relation.

Definition 4 (Implication Extraction): If $(Y \subseteq A)$, then $A \subset \gamma o\lambda(A)$ (i.e. A is strictly included in $\gamma o\lambda(A)$) then A $\Rightarrow(\gamma o\lambda(A) - A)$ is an implication extracted from the formal context (O, A, R). $(\gamma o\lambda(A) - A)$ is the additional subset of attributes co-occurring for all objects sharing attributes A. $A \subseteq \gamma o\lambda(A)$.

Illustration: We apply Galois Connection on the formal context in Fig. 1, here above, if we assume that $Attr_i$ means that, an object is divisible by i, then,

Closure$(\{Attr_1, Attr_4\}) = \{Attr_1, Attr_2, Attr_4\}$

Closure$(\{Attr_1, Attr_6\}) = \{Attr_1, Attr_2, Attr_3, Attr_6\}$

Closure$(\{Attr_1, Attr_2, Attr_3\}) =$
$\{Attr_1, Attr_2, Attr_3, Attr_6\}$

Closure$(\{\}) = \{Attr_1\}$

Consequently, we may derive the following implications:

$Attr_1 Attr_4 \Rightarrow Attr_2$;(1)

$Attr_1Attr_6 \Rightarrow Attr_2Attr_3$;(2)
$Attr_1Attr_2Attr_3 \Rightarrow Attr_6$;(3)
$\{\ \} \Rightarrow Attr_1$; (4)

Implication (4) means that any object is divisible by 1, while implication (1) means that any object (i.e. a number between 1 and 12) divisible by 1, and 4, is also divisible by 2. Implication (2) means that any object divisible by 6, and 1, is also divisible by 2 and 3. Implication (3) means that any object divisible by 1, 2, and 3 is divisible by 6. By following the line diagram of the lattice of concepts, to prove implication (1), first you search for the formal concept with the smallest intention (i.e. range) containing the premises of the rule. In this case, in formal concept $C3$ (i.e. any object of the formal context satisfying attributes $\{Attr_1, Attr_4\}$, satisfies $Attr_2$).

As the number of implications and formal concepts may grow exponentially in practice, it is not efficient to use the lattice of concepts for mining big data. In [4], we used the minimal conceptual coverage of a formal context for a new conceptual machine learning method, and in [5], we use "optimal concepts" to find the main topics in a piece of text. In this research, we used reduction algorithms that reduces a formal context without any loss of implications based on [16]. By keeping track of a reduced formal context, we guarantee that we are preserving all functional dependencies between the different features of the initial trace. In fact, in the next section, we present recent algorithms for reducing the set of objects, without losing any implication of the initial formal context. In [16], we generalized these algorithms to a fuzzy context and developed an algorithm for reducing such contexts without loss of implications within some approximation. Based on the paper published in 2014 [17], the authors found a bridge between functional dependencies in the normal database and a mapped formal context, such that any implication in the formal context is equivalent to a functional dependency in the initial database.

By using the reduction algorithm on a formal context without loss of implications, we found a minimal set of patterns that represented the initial database. Starting from these patterns, we are even able to reduce the initial database without any loss of functional dependencies.

IV. CONCEPTUAL PATTERN EXTRACTION FROM DATA

In [18], Ganter defined a clarified formal context as shown in Def. 5.

Definition 5: We say that a formal context is clarified if no two objects intentions are equal, and no two of its attributes extensions are equal.

In the next section, we give a general definition of a formal context reduction operator.

A. FORMAL CONTEXT REDUCTION OPERATOR

Definition 6 (Formal Context Reduction): An attribute m of a clarified context is said to be reducible if there is a subset of attributes $S \subseteq A$, not containing m, such that $\lambda(S) = \lambda(\{m\})$, otherwise it is irreducible. Similarly, an object o of a clarified context is reducible if there is a subset of objects

	$a1$	$a2$	$a3$	$a4$
o_1	0.5	1	0.7	0.5
o_2	0.6	0.7	1	0.5
o_3	1	0.9	1	0.1
o_4	1	0.9	0.9	0.1

FIGURE 3. Fuzzy binary context $R1$.

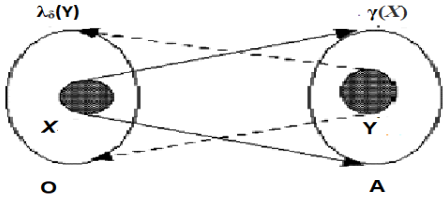


FIGURE 4. Fuzzy Galois connection schema.

$X \subseteq O$, such that $\gamma(X) = \gamma(\{o\})$. A formal context is called reduced if all of it's objects and attributes are irreducible.

In [16] we extended it to a multi-level conceptual data reduction approach for fuzzy formal contexts, based on the Lukasiewicz implication definition. "Let U be a set, called the universe of discourse. Elements of U are denoted by lowercase letters. A fuzzy set $E = \{x1/v1, x2/v2, \ldots, xn/vn\}$ is defined as a collection of elements $xi \in U, i = 1 : n$ which includes a degree of membership $vi = \mu_E(xi) \in [0, 1]$ for each element xi. The membership degree may also be expressed by a membership function.

Definition 7 (Fuzzy Binary Context): A fuzzy binary context is a fuzzy set defined on the product of two sets O (set of objects) and P (set of properties).

Example: Let us consider the fuzzy relation $R1$ depicted in Fig. 3. $R1$ contains four objects $\{o1, o2, o3, o4\}$ described by four properties $\{a1, a2, a3, a4\}$ where the values have been set randomly.

Definition 8 (Lukasiewicz Based Fuzzy Galois Connection): Let R be a fuzzy binary relation defined on U. For two sets X and Y such that $X \subseteq O$, Y is a fuzzy set defined on A, and $\delta \in [0, 1]$. We define operators γ and λ_δ, where λ_δ is generalization of λ to fuzzy context with precision δ:

$$\gamma(X) = \{d/\alpha | \alpha = min\{\mu_R(g, d) | g \in X\}, d \in A\} \quad (1)$$

$$\lambda_\delta(Y) = \{g | d \in P \Rightarrow (\mu_Y(d) \rightarrow_L \mu_R(g, d)) \geqslant \delta\} \quad (2)$$

where $\rightarrow_L$ stands for the Lukasiewicz implication i.e. for $a, b \in [0, 1]$, $a \rightarrow_L b = min(1, 1 - a + b)$.

For the objects subset X, $\gamma(X)$ is the fuzzy set of their common properties since we use the min operation. Similarly, for the fuzzy subset Y of properties,$\lambda_\delta(Y)$ computes the set of all objects that satisfy all properties in Y at a given level δ. Operators γ and λ_δ represent a fuzzy Galois connection between the subsets X and Y as illustrated in Fig. 4.

Definition 9 (Fuzzy Closure Operator): For two sets X and Y such that $X \subseteq O$, Y a fuzzy set defined on A and $\delta \in [0, 1]$. We define $Closure(X) = \lambda_\delta(\gamma(X))$ and $Closure(Y) = \gamma(\lambda_\delta(Y))$.

"Divides"	$Attr_1$	$Attr_2$	$Attr_3$	$Attr_4$	$Attr_5$	$Attr_6$
o_3	x		x			
o_4	x	x		x		
o_5	x				x	
o_6	x	x	x			x
o_{12}	x	x	x	x		x

FIGURE 5. Reduced objects in the context.

"Divides"	$Attr_1$	$Attr_2$	$Attr_3$	$Attr_4$	$Attr_5$
o_3	x		x		
o_4	x	x		x	
o_5	x				x
o_6	x	x	x		
o_{12}	x	x	x	x	

FIGURE 6. Reduced attributes in the context.

In order to simplify the concrete applications, in this paper, we map the fuzzy context with different precision levels to a crisp formal context, and then we apply the reduction operator to it. The reduction algorithm has an interesting property making it suitable to apply in a commutative or incremental manner to a formal context. If the algorithm reduces an element from a subset of the formal context, it is also removed from any set containing it, and particularly from the full context.

An attribute m of a clarified context is said to be reducible if there is a subset of attributes $S \subseteq A$,,not containing m, such that $\lambda(S) = \lambda(\{m\})$, then we may remove the attribute m, and any superset of S satisfying the same condition would have the same effect.

Example: First we clarify the context by removing duplicate objects: $\{o7, o8, o9, o10, o11\}$. For reduction, we should remove o_1, replaced by $\{o2, o3\}$ or any superset not containing o_1. We may also remove o_2, as might be replaced by: $\{o4, o6\}$ or any superset not containing o_2.

Finally, we obtain the following set of remaining objects: As a second step, we try to remove attributes: $Attr_6$ is reducible because:

$\lambda(\{Attr6\}) = \lambda(\{Attr1, Attr2, Attr3\}) = \{o6, o12\}$

This reduction might be explained by the fact that any object dividable by 2 and 3 is dividable by 6, and vice versa. In Fig. 6, we find the irreducible context.

The irreducible context, should give exactly the same implications as the initial complete context, to which we should add:

$Attr_1Attr_4 \Rightarrow Attr_2$; (1)

$\{\ \} \Rightarrow Attr_1$; (4)

To which we should add:

$Attr_1Attr_6 \Rightarrow Attr_2Attr_3$; (2)

$Attr_1Attr_2Attr_3 \Rightarrow Attr_6$; (3)

B. MAPPING FUNCTIONAL DEPENDENCIES TO IMPLICATIONS

The suggested pattern-based reduction concentrates on the representation of conceptual data. First, the input dataset of the formatted program traces is converted into a binary

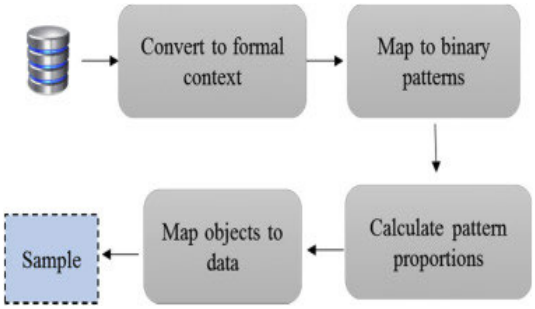
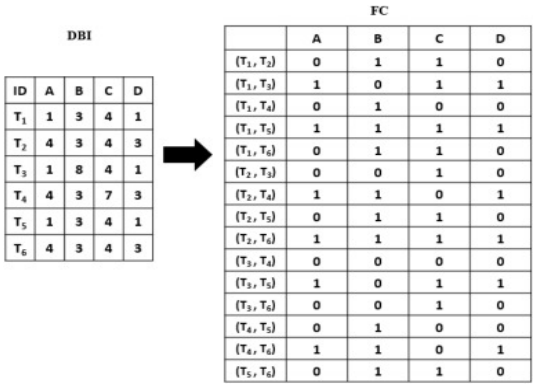


FIGURE 7. Proposed reduction method.



DBI

ID	A	B	C	D
T_1	1	3	4	1
T_2	4	3	4	3
T_3	1	8	4	1
T_4	4	3	7	3
T_5	1	3	4	1
T_6	4	3	4	3

FC

	A	B	C	D
(T_1, T_2)	0	1	1	0
(T_1, T_3)	1	0	1	1
(T_1, T_4)	0	1	0	0
(T_1, T_5)	1	1	1	1
(T_1, T_6)	0	1	1	0
(T_2, T_3)	0	0	1	0
(T_2, T_4)	1	1	0	1
(T_2, T_5)	0	1	1	0
(T_2, T_6)	1	1	1	1
(T_3, T_4)	0	0	0	0
(T_3, T_5)	1	0	1	1
(T_3, T_6)	0	0	1	0
(T_4, T_5)	0	1	0	0
(T_4, T_6)	1	1	0	1
(T_5, T_6)	0	1	1	0

FIGURE 8. Converting database instance into a formal context (set of patterns).

association declared as a formal context. This association is mapped into binary patterns and then, a part of each pattern is computed and converted back to the original data. Next Fig. 7 demonstrates the main steps. The following discussion describes these steps in more details.

1) CONVERT DATA TO FORMAL CONTEXT

Data is transformed into the formal context by conducting pairwise comparisons between data rows representing serial traces as in [7]. Fig. 8 exhibits a database instance (DBI) that is converted into a formal context FC. As an example, Rows T_1 and T_2 are compared for attributes A, B, C, and D. This comparison is translated into a different object in the formal context FC with the same attributes, where "0" is used in case the values are different, and "1" otherwise. As an example, the value $T_1(A) = 1$ is not equivalent to $T_2(A) = 4$; consequently, (T_1, T_2) (A) in the FC is "0". However, $T_1(B) = T_2(B)$, and therefore, (T_1, T_2) (B) in FC is "1". Every row is also compared to itself giving a same pattern containing only ones ("1").

One disadvantage of this transformation approach is the precise matching between numbers, which causes information loss. To overcome this problem, a similarity measure is identified to process the pairwise comparison process as discussed in [33]. It is computed as follows: Similarity $(n_1, n_2) = ((1 - (|n_2 - n_1|)/\max(n_2, n_1))$, where n_1, n_2 are the two numbers to compare. If two numbers are equal then they are similar with a degree 1. Numbers $(3, 4)$ are less similar than $(1000, 1001)$, while they both differ by the same value 1. The reason is that relatively to the maximum

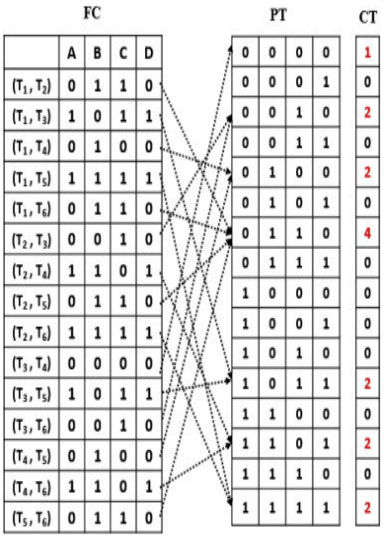


FIGURE 9. Mapping *FC* objects to patterns.

of the two values, 1000 and 1001 are judged more similar to each other than 3 and 4. Depending on the previous rule, the percentage of similarity is measured through the pairwise comparison. In case the similarity between the compared data values is greater than a defined threshold, then the FC object value is "1". Otherwise, the value of the FC object is "0".

2) MAP A DATABASE TO BINARY PATTERNS

From the format context FC, we generate the binary patterns. The number of features existing in FC is equal to the number of features in the dataset.

As an example, with a format context having M attributes, we obtain a binary pattern table of size 2^M patterns, where each tuple of the formal context is associated to one of these patterns. A counter is attached with each binary pattern, and it is incremented whenever an object of the FC is mapped with this pattern. A specific number of FC objects is preserved per pattern to reduce the memory consumption. Fig. 9 depicts an example of a binary pattern table (PT) and the counter table (CT) which are generated from FC with four attributes. As such, since objects (T_3, T_6) and (T_2, T_3) are associated with the pattern "0010", its corresponding counter value in the CT will be 2.

V. METHODOLOGY FOR ANOMALY DETECTION

In this section, we start by explaining our investigation concerning how to discover anomalies by using a formal context that abstracts the knowledge about generated traces in the program. After a preliminary analysis of generated traces, in section V-A, we develop our proposed method for detecting anomalies in a simple module by employing a learning step. In section V-B, we generalize it for complex software and for different number of features.

For anomaly detection, we start by saving the program traces for a specific input I into a database T. Selected program state variables are used to represent the database features. Any anomaly occurrence in the trace is reflected as an anomaly in T. Therefore, any used method for detecting

X	Y
9	7
10	6
11	5
12	4
13	3
14	2
15	1
16	0

FIGURE 10. Execution trace (*ET*).

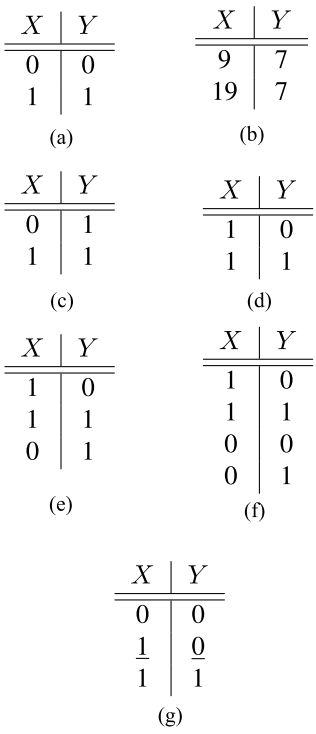


FIGURE 11. (a) Reduced formal context (i.e. patterns). (b) Abnormal program states. (c) Abnormal patterns- case 1. (d) Abnormal patterns- case 2. (e) Abnormal patterns- case 3. (f) Abnormal patterns- case 4. (g) Anomaly detected in function *Sum*.

abnormal instances in a database becomes directly applicable for programs.

Example: Assume that a program, should calculate the sum of two numbers $X = 9$, $Y = 7$, then a possible algorithm is to increment X by 1 and decrease Y by 1, until the value of Y becomes equal to 0, and getting the sum in X:

The execution trace ET should be as in Fig. 10.

Traces in Fig. 10 are mapped to a formal context of patterns of Fig. 11a.

From patterns in Fig. 11a, we find the only implications: $X \implies Y$ and $Y \implies X$.

An anomaly is detected as soon as we find a new pattern that could not be reduced to the correct ones in Fig. 11a.

Traces of Fig. 11b are not correct. We can see in Fig. 11c, the corresponding abnormal patterns containing row $(0, 1)$. In fact, we may only conclude implication $X \implies Y$, but not $Y \implies X$.

In Fig. 11d, we get an abnormal formal context corresponding to ($Y \implies X$).

We may even obtain the cases of no dependencies, as in Fig. 11e or Fig. 11f.

For the sum function, when we inject errors that contaminate the critical information of the program, we got the patterns of FC outputs as in Fig. 11g. Critical information is an invariant condition that the state of the program should preserve for a correct execution. In the case of function *Sum*, $X + Y$ should remain unchanged at the beginning of each iteration.While increasing X by 1, Y is decreasing by 1, as shown in Fig.10. We may notice that row (1 0) does not belong to the correct FC pattern mentioned as in Fig. 11g.

If ($X \geq 0$ and $Y = 0$) we only obtain the row (1 1) included in the correct FC patterns.

A. A GENERAL ANOMALY DETECTION ALGORITHM

Anomaly detection is modular. Each module in a complex piece of software should go through two stages; first is the learning phase which generates some knowledge K describing regular patterns (see section V-A.1), and second is the regular utilization of K to discover some anomalies at run-time of the module, and in turn, alert owners about them (see section V-A.2). In section V-B, we give a general description of the method in the context of software engineering life cycle, and for complex software.

1) THE LEARNING PHASE

The learning phase involves a similar process for each module in the program. First, we should design the features for module tracing to be informative enough about the state of the program: it should contain whether the input state changed during execution or not, working variables that might include some input variables, and incremental solution construction as in most dynamic programming algorithms. Checkpoints used to generate the trace should reflect a state change, or invariant control. During this step, we run the current module M, using a random data set T, requiring no human effort. For each input test t belonging to T, we run M, generate the list Tr of several traces, and transform it to a reduced formal context FCt containing all dependencies between the different features of Tr. Finally, by compiling incrementally all FCt, for all traces t in T, we generate the learnt knowledge database K. The learning step is applied for correct executions of each module M. During the learning step; we merge all generated patterns corresponding to all inputs into one set in reduced form. This sample then becomes our reference for detecting anomalies in the future regular life cycle of the module.

Let F be the selected set of tracing variables (i.e. features of the trace) of a module M. Algorithm 1 shows the learning step.

After a sufficient number of module runs, having combined multiple patterns to set K, it reaches some stability. Here, stability is measured as the percentage of non-reduced patterns with respect to the current size of K (i.e. reflecting a form of saturation). This constitutes an excellent criteria to assess the learning process completeness. In the sorting algorithm example that we use to evaluate our method, we are able to reach 100% saturation with only a few random input selections.

Algorithm 1 Learning Step Algorithm for Building Knowledge Database of Patterns K for Detecting Anomalies in a Given Module M

Input: a Module M With Input Size n
Output: the Knowledge Dataset of Patterns K

```
K = empty set;
while K is still not stable do
    Let S a new random input of size n;
    while not end of execution of module M do
        Generate Execution Trace Tr of M for input S ;
        Add formal context patterns (FC) corresponding
        to Tr into K, using a reduction approach for
        checking consistency;
    end while
end while
```

Below are several examples about identifying the features of the traces for some programs:

- In the module calculating the sum of two integers A and B, the trace is featured by the set F including the two variables X, and Y. Variable X, (respectively Y) contains initially the values that the program should add (i.e.respectively A, and B). X and Y are also the only used variables in the program starting from the inputs (i.e. initial values of X and Y).
- For searching for the position of a value x in an array A of size n, set trace feature F is composed of the input array A, the searched value x, and any browsing index. In fact, invariant as array A, and value x, or any constant in a program represent the most critical information in a running program. In case of linear search, the index I is used for comparing in each iteration $A[I]$ with x. For binary search, we add the lower and upper bounds limiting the search range.
- If we use an iterative or recursive binary search, the set of features F should contain the input A, the search value x, and the indexes, low and high bounding the search domain in A. Fortunately, as constants and input A are invariant, during the learning step, they do not increase the number of patterns, but they are essential to detect anomalies.
- In sorting algorithms of an array B of size n, we change B by consecutive swapping of its elements until obtaining a sorted array B. Set F should contain all positions of the array (i.e. $B_0, B_1, \ldots B_9$, if the size of array B is 10), a check of the right swap of two positions x and y using attribute S in Table 1, is added to the feature. In fact, if all swaps in an array are correctly executed, then "any array state is always a permutation of the initial array B": this

condition is not sufficient, but it is necessary for program correctness.

- In the case of Kruskal's greedy algorithm for extracting a minimum spanning tree in a graph $G = (E, V)$, a trace is generated after each selection of an edge from the graph. The feature F of the trace contains the weight of the selected edge, the number of edges in variable e, the number of nodes in variable V, and selected pairs of vertices so far. Weights of edges should not change in the program and their loss by contamination is fatal. As Kruskal algorithm invokes function sorting of the list of edges in non-decreasing order of their weights, the control of anomalies should be done separately for the sorting step.

In general, traces should be generated, mostly inside repetitive statements at each potential change of at least one field of trace features. As examples, for adding two numbers X and Y, using only an increment or a decrease by one, we generate a trace each time we execute an iteration in the loop.

For linear search algorithm, we generate a trace after each increase of an index by one. For binary search, we generate a trace after each update of either the lower or the higher bound of the searching range. For sorting, we trace after swapping any two elements of the array to sort.

Table 1 shows knowledge set K for Quicksort, obtained incrementally from different FCs generated from training inputs. In fact, the number of patterns for sorting algorithms built that way, should be mathematically 46. This is what we obtained after only 10 random inputs, as by then, the set K becomes stable.

2) DETECTING ANOMALIES

During the normal software life cycle, we discover an anomaly each time we generate an unknown pattern not reducible by knowledge K built during the learning step. A trace is generated of a running module for some input and a formal context FC is created from it. If a generated pattern of FC is not reducible by K then it is considered as abnormal. Users are alerted at real time about it.

In Table 2, we may see an example of Correct Formal Context corresponding to mined traces of Quicksort algorithm. We also notice that in each row we have only two "0", or all "1" for one case. This means that any row with more than two "0" is reduced, because it may be replaced by a concept containing only rows with two "0". This might be explained logically as well: in fact, pattern (1-2) reflects a swapping of two elements in the trace 1 giving trace 2. Pattern (2-3) reflects a swapping of two elements in the trace 2 giving 3, then pattern (1,3) represents the two consecutive swaps. Together patterns (1-2) and (2-3) represent the concept corresponding to (1,3) because it included all "1"s in (1-2) and (2-3) only.

In each row, we only have exactly two zeros or all ones in the last row of the Table 2. By generalization, as we have 10 elements in the array, we have 45 different possibilities to have only two zeros in a pattern, and one pattern with all features set to one. Thus, with an array of size n, we have $(n(n-1)/2+1)$ total possible acceptable patterns after reductions. So all other patterns should either be reduced or correspond to some anomaly. Therefore, the program has some anomalies that we could detect online each time the two compared traces give a non-reducible pattern with more than two zeros or with only one zero. This result, may be applied to any sorting algorithm, where a trace is generated each time we swap two elements.

When the size $n = 10$, as the last column S is always set to one, then we may code each pattern by only by the 10 first columns from (B_0 to B_9). The 46 binary patterns

TABLE 1. Knowledge K for quicksort.

B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	S
1	0	1	1	1	1	0	1	1	1	1
1	1	0	1	1	1	1	1	1	0	1
0	0	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	0	1	1
1	1	1	1	1	0	1	0	1	1	1
1	1	1	1	1	1	1	0	1	0	1
1	1	1	0	1	1	0	1	1	1	1
1	1	1	1	0	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0	0	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	0	1	1	1	1
1	1	1	1	1	1	0	1	0	1	1
0	1	1	1	0	1	1	1	1	1	1
1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	0	0	1	1	1	1	1
1	1	1	1	1	1	1	0	0	1	1
0	1	1	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	0	1	1	1
1	1	1	1	1	0	1	1	1	0	1
1	1	1	0	0	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	0	1	1
1	1	1	1	1	1	0	1	1	0	1
1	1	0	0	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	0	1
1	1	1	1	1	1	0	0	1	1	1
1	0	1	1	1	1	1	1	1	0	1
1	1	1	1	1	0	1	1	0	1	1
1	1	0	1	0	1	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	1	1	0	1	1	1	1	1	1
1	1	1	1	0	1	1	0	1	1	1
0	1	1	1	1	1	1	0	1	1	1
1	1	0	1	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	0	1
1	1	0	1	1	1	1	1	0	1	1
1	1	0	1	1	0	1	1	1	1	1
1	0	1	0	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	0	1	1
1	1	1	0	1	1	1	0	1	1	1
0	1	1	1	1	0	1	1	1	1	1

TABLE 2. Reduced formal context corresponding to correct quicksort traces.

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	S
(1-2)	1	1	1	1	1	0	0	1	1	1	1
(2-3)	1	1	1	1	1	1	0	1	0	1	1
(3-4)	0	1	1	1	0	1	1	1	1	1	1
(4-5)	1	1	1	0	1	0	1	1	1	1	1
(5-6)	0	0	1	1	1	1	1	1	1	1	1
(6-7)	1	1	1	1	0	0	1	1	1	1	1
(7-8)	1	1	1	1	1	1	1	0	0	1	1
(8-9)	1	1	1	1	1	1	1	1	1	1	1

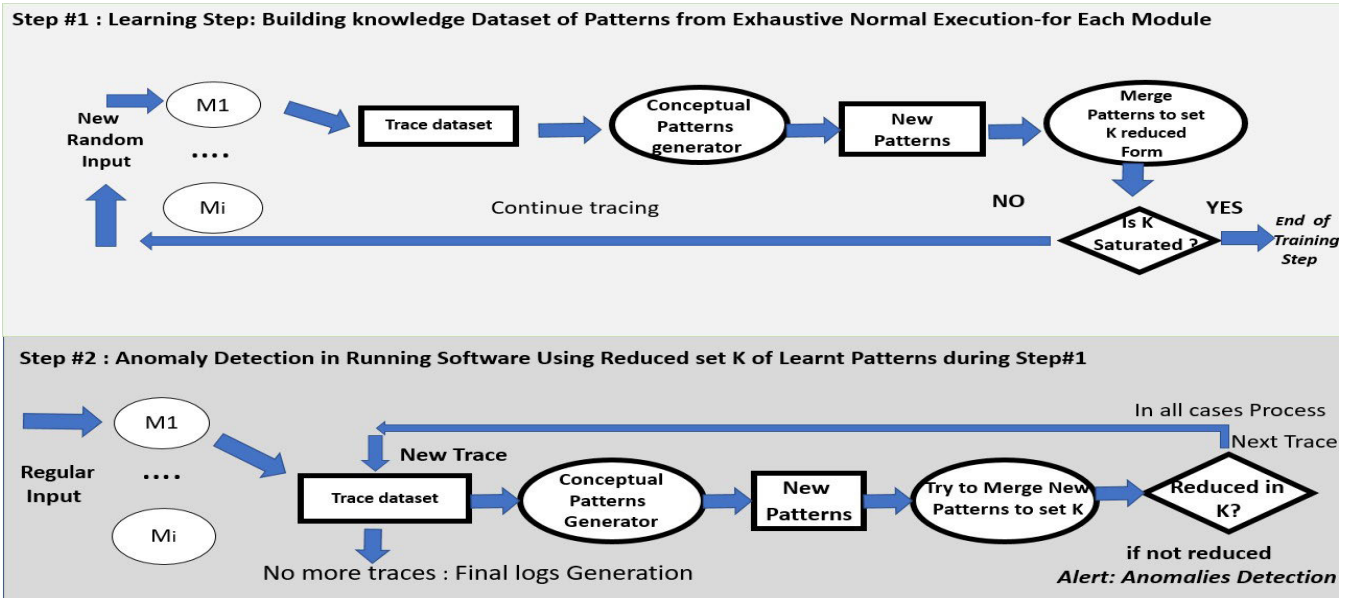


FIGURE 12. General schema about the two steps for learning and detecting anomalies for each module.

TABLE 3. Reduced formal context corresponding to contaminated quicksort traces.

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	S
(1-2)	1	1	1	1	1	0	0	1	1	1	1
(2-3)	1	1	1	1	1	1	0	1	0	1	1
(3-4)	0	1	1	1	0	1	1	1	1	1	1
(4-5)	1	1	1	0	1	0	1	1	1	1	1
(3-6)	0	0	0	0	0	0	0	1	1	1	0
(6-7)	1	1	1	1	1	1	1	0	0	1	1
(7-8)	1	1	1	1	1	1	1	1	1	1	1

converted to decimal numbers correspond to the following set of codes:

Authorized patterns = {255, 383, 447, 479, 495, 503, 507, 509, 510, 639, 703, 735, 751, 759, 763, 765, 766, 831, 863, 879, 887, 891, 893, 894, 927, 943, 951, 955, 957, 958, 975, 983, 987, 989, 990, 999, 1003, 1005, 1006, 1011, 1013, 1014, 1017, 1018, 1020, 1023}.

Represented using 10 bits, each number matches a string of only two bits equal to zeros or all bits are one (i.e. 1023). This means that each time we generate a pattern coded by decimal number P, we only need to check that P belongs to the set of authorized patterns, that we could do in about the ceiling of $log_2(46) = 6$, and that feature S is equal to 1. When we inject errors by damaging the array during program run time, the following formal context in the Table 3 generated from the new traces shows an anomaly in row (3,6) with more than two ''0''. This might be implemented easily during pattern generation. We can say that we learnt from normal formal context as in Table 1, how to detect anomalies for non-similar generated formal contexts as in Table 3.

In the current subsection, we presented some illustrations of how our method runs. In the next subsection, we will present the general methodology for anomalies detection of a program during run time.

B. THE GENERALIZATION OF ANOMALY DETECTION

In this section, we describe our approach from the software engineering life cycle perspective. The general method for anomalies detection is as structured as the software itself in a modular way. We can see in Fig. 12 the composition of the two different cycles: Learning step & Anomalies detection step. For each module of a given class, we extract all needed features for tracing it during it's run-time. These features are mainly composed of inputs, outputs, and final constant values such as the size of an array, the number of nodes in a graph, or the value of an input that we are searching in a list, the number of days in April, or the days of the week if involved in the module. If the number of features is too high, it is recommended to select a limited number of features. This is important in order to get a solution that does not affect the program performance heavily. For each module, we perform the following actions for the learning step:

- Trace feature selection with limited size. Adequate features extraction should be designed carefully by the software designers. In fact, if a module calls another one, input arguments defined by the caller are generally included in the feature of the called module.
- Run the module with random inputs.
- For each trace, generate the corresponding formal context FC.
- Reduce FC into the global consistent K of patterns using the same reduction algorithm, Integrating anomaly checking in the module.
- Incrementally generate traces with the same selected features as in the learning step for a new input.
- Generate the corresponding formal context FC incrementally during run-time, if a pattern in FC is not reduced by the global knowledge K, we alert with an anomaly.

VI. VALIDATION OF THE METHOD AND ITS LIMIT

The presented method is limited to only finding contamination related to critical information in the program i.e. the part of code responsible for the loss of essential data is detected. For example, the method detects anomalies caused by the

TABLE 4. Experimental results of *Sum* and *Quicksort*.

Program	Injected errors	False anomalies	Correct Anomalies	Correct normality	False Normality
Sum of two integers	NO error injected	0%	N/A	100%	0%
Sum of two integers	while((x!=0)){x–; if(y%2==0) y++; else y–; count++; }	0%	96%	N/A	4%
Sorting Algorithm: *quicksort*	NO error injected	0%	N/A	100%	0%
Sorting Algorithm: *quicksort*	temp=s[rightInd]; s[rightInd]=s[leftInd]; \\ s[leftInd] =temp;\\ removed statement → wrong swapping Also destroying values in the array: for (int i = 0; i < 7; i++) { int z = r.nextInt(100); s[i] = new Integer(i*7+z);}	0%	81.4%	N/A	18.6%
Sorting Algorithm: *quicksort*	\\ s[leftInd]=temp; \\ removed statement → \\ wrong swapping	0%	100%	N/A	0%
Sorting Algorithm: *quicksort*	for (int i = 0; i < 7; i++) { int z = r.nextInt(100); s[i] = new Integer(i*7+z);} \\ Overwriting some values \\ of the array	0%	100%	N/A	0%

contamination of elements in an array during the sorting process, or the graph weights in the case of the minimum spanning tree program. However, as we assume that the program is correct, we cannot detect whether it is well designed or not, or if the solution is correct. In such cases, we might obtain a false positive due to some algorithm design deficiency. In fact, as a limit of the proposed approach, we only try to find anomalies about the program states through some selected features. We systematically obtain an alert in case of any sudden change in the controlled invariant data or in the list of saved partial solutions. As we observed in Table 4, in all tested cases, when we do not inject errors in the programs *Sum* or *Quicksort*, no anomaly is detected. When we inject errors contaminating the invariant in program *Sum*, we got 96% of anomalies detected. For *Quicksort*, incorrect swapping of two elements in an array gives 81.4% of real anomaly detected. If some values in the array are overwritten, we detect 100% of these anomalies. Added to the originality to the proposed learning method compared to other machine learning methods, it is giving very high accuracy in general. In [1], even not applied with our same tested programs, the learning-based run-time anomaly detection in software systems detects 70% of anomalies.

We could not observe a code or data change if it does not contaminate the state of the program. In that case the program might end with a correct output, as for *Kruskal* algorithm when the selected edge is not creating a cycle. The method only detects a change in the behavior of the program provoked by a code change or data loss. No alert occurs if the contamination makes a change in the algorithm which is not reflected in the generated patterns. Through realized experience, with

TABLE 5. Experimental results of *Kruskal* 1.

Program	Injected errors	False anomalies	Correct Anomalies	Correct normality	False Normality
Kruskal Algorithm	No injected errors	0%	N/A	100%	0%
Kruskal Algorithm	if((l!=k)&&edge[k].used) writeOnCsvFile("1"); else writeOnCsvFile("0"); \\inversing the condition (l==k) to (l!=k)	0%	100%	N/A	0%
Kruskal Algorithm	writeOnCsvFile (Integer.toString(1));u=k+1; \\Injected error: Changed 0 by 1	0%	100%	N/A	0%
Kruskal Algorithm	if(result[u].src==j) {writeOnCsvFile (Integer.toString(0)); u=k+1;} elseif(result[u].dest==j) {writeOnCsvFile (Integer.toString(0)); u=k+1;} \\1 changed by 0 here underlined.	0%	100%	N/A	0%

TABLE 6. Experimental results of *Kruskal* 2.

Program	Injected error	False anomalies	Correct Anomalies	Correct normality	False Normality
Kruskal Algorithm	Injected error: {writeOnCsvFile (Integer.toString(1)); u=k+1;} else if(result[u].dest == j) \\0 instead of 1 {writeOnCsvFile (Integer.toString(0));u=k+1;} elseif(u==k) {writeOnCsvFile (Integer.toString(0)); u=k+1;}} //1 changed by 0 underlined.	0%	96%	N/A	4%
Kruskal Algorithm	Remove the check of cycles. (if (x!=y), Over 100 tests , 38 passed correctly in spite of check cycle removal, Only 14 over 62 abnormal cases passed (23%). Total of correct decisions = 52% correct decisions, for 100 random tests.	0%	23%	100%	77%
	After Improvement by invariant control check addition to the trace feature, we got a much better results of 100% true anomalies detected (around 73%). The error is not detected, but it is naturally recovered giving correct output (i.e. minimal spanning tree)	0%	100%	100%	0%

high probability, we recognize a non-reducible pattern in the proximity of the injected errors either in the code or in the data.

In Table 5 , for *Kruskal* algorithm, we got very good accuracy (100%) with no injected errors, and 100% when we reversed some conditions. We can also see in Table 6 that accuracy is 96% when we incorrectly changed some values from 0 to 1 in partial results (i.e. selected edges in the spanning tree). In the last case, when we removed the condition checking if the selection of a new edge creates a cycle or not, and we did not add some invariant verification in the trace, we got only 23% of true anomalies detected. Fortunately, after inclusion of invariant preservation tracing, we got 100% of anomalies detected. Generally in all cases, the anomalies are recognized almost immediately after the damaged code or data. Anomalies detection method may be easily applied for any software in a systematic way, and give very good accuracy.

VII. CONCLUSION

In this paper, a novel method is presented for early discovery of program execution anomalies after learning from "initially correct" program executions. Such anomalies could be introduced by malicious or inadvertent code change, implementation bugs and software updates that might provoke regression errors, code security issues, and fatal or transient faults. The program starts by building conceptual patterns (i.e. knowledge K) as a reduced union set of formal concepts FCs generated from the program traces applied to random input sets. The process of building the patterns continues until stability is reached. Thereafter, the learned knowledge K is used throughout the program life cycle to find anomalies during execution. Validation of the method was performed on several well-known algorithms. The results demonstrate excellent accuracy either for detecting the normality of the program execution or anomalies when analysing the traces generated during program execution.

The modular approach introduced during the software life cycle gave encouraging results by successfully detecting several categories of errors. The design of the trace features for each module, containing variables representing its inputs, outputs, and intermediate working variables is important for accurate anomaly detection. We observed that the most important features to include are program invariant checks and permanent persistent data in the program. This study gives the designer another view about the software while preparing for featured trace generation. It links tracing to the software goals.

It is worthwhile for future work to focus on extending this research to the case of large features, corresponding to complex software with higher number of inputs and outputs.

REFERENCES

[1] F. Huch, M. Golagha, A. Petrovska, and A. Krauss, "Machine learning-based run-time anomaly detection in software systems: An industrial evaluation," in *Proc. IEEE Workshop Mach. Learn. Techn. Softw. Qual. Eval. (MaLTeSQuE)*, Mar. 2018, pp. 13–18.

[2] K. Böhmer and S. Rinderle-Ma, "Mining association rules for anomaly detection in dynamic process runtime behavior and explaining the root cause to users," *Inf. Syst.*, vol. 90, May 2020, Art. no. 101438.

[3] L. Zemín, S. G. Brida, S. Bermúdez, S. P. D. Rosso, N. Aguirre, A. Mili, A. Jaoua, and M. F. Frias, "Stryker: Scaling specification-based program repair by pruning infeasible mutants with SAT," 2019, *arXiv:1910.14011*. [Online]. Available: http://arxiv.org/abs/1910.14011

[4] X. Cheng, "Anomaly detection and fault localization using runtime state models," M.S. thesis, Elect. Comp. Eng., Univ. Waterloo, Waterloo, ON, Canada, 2016.

[5] M. Maddouri, S. Elloumi, and A. Jaoua, "An incremental learning system for imprecise and uncertain knowledge discovery," *Inf. Sci.*, vol. 109, nos. 1–4, pp. 149–164, 1998.

[6] E. Rezk, Z. Awan, F. Islam, A. Jaoua, S. Al Maadeed, N. Zhang, G. Das, and N. Rajpoot, "Conceptual data sampling for breast cancer histology image classification," *Comput. Biol. Med.*, vol. 89, pp. 59–67, Oct. 2017.

[7] E. Rezk, S. Babi, F. Islam, and A. Jaoua, "Uncertain training data set conceptual reduction: A machine learning perspective," in *Proc. IEEE Int. Conf. Fuzzy Syst. (FUZZ-IEEE)*, Jul. 2016, pp. 1842–1849.

[8] S. Al-Maadeed, F. Ferjani, S. Elloumi, and A. Jaoua, "A novel approach for handedness detection from off-line handwriting using fuzzy conceptual reduction," *EURASIP J. Image Video Process.*, vol. 2016, no. 1, p. 1, Dec. 2016.

[9] C. A. Hoare, "Quicksort," *Comput. J.*, vol. 5, no. 1, pp. 10–16, 1962.

[10] G. Michael and J. David, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman and Company, 1979.

[11] Y. Chen, M. Ying, D. Liu, A. Alim, F. Chen, and M. Chen, "Effective online software anomaly detection," in *Proc. Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 2017, pp. 136–146.

[12] L. Jian-Guang, F. Qiang, Y. Shengqi, X. Ye, and L. Jiang, "Mining invariants from console logs for system problem detection," in *Proc. USENIX ATC*, 2010, pp. 231–244.

[13] J. F. D. Addison, S. Wermter, and J. MacIntyre, "Effectiveness of feature extraction in neural network architectures for novelty detection," in *Proc. 9th Int. Conf. Artif. Neural Netw. (ICANN)*, Sep. 1999, pp. 976–981.

[14] K. Bsaïes, F. Hammami, A. Jaoua, and W. Ksontini, "May reasoning be reduced to an information retrieval problem," in *Proc. Relational Methods (RelMiCS)*, Warsaw, Poland, 1998, pp. 29–32.

[15] B. Ganter, G. Stumme, and R. Wille, Eds., *Formal Concept Analysis: Foundations and Applications* (Lecture Notes in Artificial Intelligence), vol. 3626. Berlin, Germany: Springer-Verlag, 2005.

[16] S. Elloumi, J. Jaam, A. Hasnah, A. Jaoua, and I. Nafkha, "A multi-level conceptual data reduction approach based on the Lukasiewicz implication," *Inf. Sci.*, vol. 163, no. 4, pp. 253–262, Jun. 2004.

[17] J. Baixeries, M. Kaytoue, and A. Napoli, "Characterizing functional dependencies in formal concept analysis with pattern structures," *Ann. Math. Artif. Intell.*, vol. 72, nos. 1–2, pp. 129–149, Oct. 2014.

[18] B. Ganter, G. Stumme, and R. Wille, "Formal concept analysis: Methods and applications in computer science," Dept. Math., TU Dresden, Dresden, Germany, Tech. Rep., 2003.

AHMAD QADEIB ALBAN is currently pursuing the master's degree in computer science with Qatar University (QU). Besides, he is a Research Assistant with the QU on a granted-research project which aims at the development of a new secure computer model called the garbled computer (GC) as well as detecting the anomalies that may occur in the runtime of the code. His research interests include cybersecurity, cloud computing, machine learning, and deep leaning as well as distributed systems. He is also interested in blockchain and edge data structures.

FAHAD ISLAM received the B.Sc. degree in computer science from Carnegie Mellon University. He has worked as a Research Assistant with Qatar University on a granted research project that focuses on extracting analytical information from deep-web databases and automatically constructing new interfaces to assist the user with their search process. His primary research experience is in the fields of formal context analysis and data analysis through machine learning. He has also experience with software development in various environments such as the GPGPU, Web, and mobile.

QUTAIBAH M. MALLUHI (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from the KFUPM, Saudi Arabia, and the M.S. and Ph.D. degrees in computer science from the University of Louisiana, Lafayette. He was the Head of the Department from 2006 to 2012 and the Director of the KINDI Center for Computing Research at the QU from 2012 to 2016. He is a Professor with the Department of Computer Science and Engineering, Qatar University (QU). His experience includes serving as a Professor with Jackson State University and a Research Faculty with the Lawrence Berkeley National Laboratory. He was the Co-Founder and the CTO of Data Reliability Inc. He has received several honors and awards including the QU Research Award, the JSU Technology Transfer Award, the Mississippi MURA, and the JSU Faculty Excellence Award.

ALI JAOUA received the Engineering degree in computer science from the ENSEEIHT, Toulouse, in 1977, and the Dr.Eng. and Ph.D. degrees in computer science from the University Paul Sabatier of Toulouse, France, in 1979 and 1987, respectively. He was a Professor in computer science at Qatar University from 2000 to 2019, where he has been an Adjunct Professor/Researcher since 2019, was an Associate Professor in computer science at Laval University, Canada, from 1989 to 1992, and a Professor in computer science at Al-Manar University, Tunisia, from 1995 to 2008, and an invited professor at different universities and conferences. He was a coordinator of graduate programs for the last six years. He has published 50 articles in international journals, presented about 100 conferences, and contributed to several books, and is a leader of several granted research projects. His main research domains are data science and software engineering. He is a member of the ACM Society.