Received June 4, 2020, accepted June 16, 2020, date of publication June 22, 2020, date of current version July 9, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3004199

Proof Learning in PVS With Utility Pattern Mining

M. SAQIB NAWAZ[1], **PHILIPPE FOURNIER-VIGER**[1], **AND JI ZHANG**[2], **(Senior Member, IEEE)**
[1]School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China
[2]School of Sciences, University of Southern Queensland, Toowoomba, QLD 4350, Australia

Corresponding author: Philippe Fournier-Viger (philfv8@yahoo.com)

ABSTRACT Interactive theorem provers (ITPs) are software tools that allow human users to write and verify formal proofs. In recent years, an emerging research area in ITPs is proof mining, which consists of identifying interesting proof patterns that can be used to guide the interactive proof process in ITPs. In previous studies, some data mining techniques, such as frequent pattern mining, have been used to analyze proofs to find frequent proof steps. Though useful, such models ignore the facts that not all proof steps are equally important. To address this issue, this paper proposes a novel proof mining approach based on finding not only frequent patterns but also high utility patterns to find proof steps of high importance (utility). A proof process learning approach is proposed based on high utility itemset mining (HUIM) for the PVS (Prototype Verification System) proof assistant. Proofs in PVS theories are first abstracted to a computer-processable corpus, where each line represents a proof sequence and proof commands in proof sequences are associated with utilities representing their weightage (importance). HUIM techniques are then applied on the corpus to discover frequent proof steps/high utility patterns and their relationships with each other. Experimental results suggest that combining frequent pattern mining techniques, such as sequential pattern mining and high utility itemset mining, with proof assistants, such as PVS, is useful to learn and guide the proof development process.

INDEX TERMS Frequent patterns, high utility itemset mining, proof steps, proof sequences, PVS.

I. INTRODUCTION

Theorem proving is a famous approach in formal methods that is used for the analysis of hardware and software systems, especially safety critical systems. In theorem proving, the system that needs to be analyzed is first modeled and specified in an appropriate mathematical logic. Important/critical properties of the system are then verified using theorem provers [21]. Today, these mathematics-based tools are used in verification projects that range from compilers, operating systems and hardware components to prove the correctness of large mathematical proofs [31]. There are two general categories of theorem provers: Automated theorem provers (ATPs) and interactive theorem provers (ITPs). ATPs are generally based on first-order logic (FOL) and deal with the development of computer programs that can automatically perform logical reasoning. On the other hand, ITPs are based on higher-order logic (HOL) and offers support for rich logical formalisms such as dependent and (co)inductive types as well as recursive functions. This expressive power leads to the undecidability problem, which means that the reasoning process cannot be automated in HOL. Thus, ITPs require human guidance during the proof development process, which is why ITPs are also known as proof assistants. Some famous proof assistants are HOL4 [39], Coq [6] and PVS [36].

In ITPs, user guides the proof process by providing the proof goal and by applying proof commands and tactics to prove the goal. Generally, a user is involved in lots of repetitive work while verifying a nontrivial theorem (proof goal), and thus the overall process is quite laborious and tedious as well as time consuming. For example, a list of 100 mechanically verified mathematical theorems is available [44]. The development of formal proofs for many of these theorems required several months or even years (For example, the Kepler conjecture proof in HOL Light [20] take approximately 20 years and twice as much for the Feit-Thompson theorem in Coq [19]) and the complete proofs contain thousands of low-level inference steps.

In the automated deduction field, huge search spaces are generally involved in finding the correct proofs for a given theorem/lemma. This means that proof automation

The associate editor coordinating the review of this manuscript and approving it for publication was Jerry Chun-Wei Lin.

in theorem provers is not as advanced yet as it was once thought they would be by this time. The formal proof of a goal in ITPs mainly depends on the specifications available in a theory or a set of theories along with different combinations of proof commands, inference rules, intermediate states and tactics. Because a theory in ITPs often contains many definitions and theorems, it is quite inefficient to apply a brute force or pure random search-based approach for proof searching. However, ITPs now do have a large corpora of computer-understandable formalized knowledge [7], [23], [24] in the form of proof libraries.

In PVS, proof scripts for each theory are stored in a separated file. Such proof scripts can be considered as a proof corpus for the theorems and lemmas in that theory. Proof scripts of different theories can be combined together to develop a more complex corpora. With the evolution in information and communication technologies (ICT) in the last decade, it is now possible to use data mining and machine learning techniques on these corpora for guiding the proof search process, for proof automation and for the development of proof tactics/strategies, as indicated in the works done in [10], [18], [22]–[24], [34].

In these corpora, there is the potential to identify useful and interesting hidden proof patterns and relationships of such proof patterns with each other. With such information, sequential pattern mining (SPM) techniques [15] can be used to investigate the dependency of new conjectures on already proved facts and to predict the next proof step(s) or pattern(s) for guiding the proof of a new non-trivial theorem/lemma. One such proof process learning approach was provided in [34]. However, the learning approach was not considering the importance of particular proof commands in the corpus. Every PVS proof command was given the same importance. However, in most cases, this is not the general way. For example, in most of the proofs, proof commands for skolemization (such as *skosimp*, and *skolem*!), formulas simplification (such as *flatten* and *split*) and definitions expansion (such as *expand*) are used at the start and powerful decision procedure such as *grind* and *assert* (for equational reasoning) are used at the end. This means that some proof commands and decision procedures are used more and should be given more importance than other infrequently used proof commands. The focus in this work is on proof guidance and premise selection in PVS from the perspective of high utility pattern mining (HUPM) techniques, particularly high utility itemset mining (HUIM).

In this paper, we present HUIM-based proof process learning approach for the PVS proof assistant. The basic idea is to convert the PVS proofs for a theory into a proof corpus that is suitable for learning. Each line in the corpus represents a proof sequence that comprises a set of proof commands. Each proof command is given a utility value that represents the importance of that proof command. HUIM techniques are applied on the corpus to find frequent proof steps/patterns with high utility that are used in the proofs. Moreover, relationships between proof steps/patterns are discovered through sequential rule mining. Besides PVS, the proposed approach can also be used to guide the proof process in other proof assistants. For example, the learning approach [34] is recently used in [35] for proof searching and optimization in the HOL4 proof assistant.

Machine learning and data mining are mainly used in theorem provers for three tasks: premise selection, strategy selection and internal guidance. Deep learning techniques were first used in [22] for premise selection in Mizar prover. For ITPs, a large datasets called *HolStep* was introduced in [23], which consists of 2M statements and 10K conjectures from HOL Light proof assistant. In [24], various machine learning methods were used and compared for learning the dependencies in proofs taken from *CoRN*, which is a repository for the Coq proof assistant. Premise selection based on machine learning and automated reasoning for the HOL4 is provided in [17] by adapting HOL(y)Hammer [25]. Moreover, Tactictoe, a learning approach based on the Monte Carlo tree search algorithm, was developed in [18] for HOL4.

The rest of this paper is organized as follows. Section II provides a brief overview on PVS, SPM and HUIM. Section III elaborates the HUIM-based learning approach that is used to discover useful proof steps/patterns with high utility and their relationships. Details on HUIM algorithms used in this work are presented in Section IV. Evaluation of the proposed approach on a case study and obtained results are discussed in Section V. Finally, the paper is concluded with some remarks in Section VI. PVS dump files (that contain entire specifications for theories and associated proofs) and HUIM related data for this work can be found at [38].

II. PRELIMINARIES

A brief introduction to the PVS proof assistant, SPM and HUIM is provided in this section.

A. PVS

Prototype verification system (PVS), developed at SRI International, offers a formal specification language and an interactive theorem prover. The PVS specification language is build on HOL and its type system supports predicate sub-typing and other type dependencies. The PVS type system is not algorithmically decidable and theorem proving may be required to establish the type-consistency in a specification. Theorems that need proving are called type-correctness conditions (TCCs). Specifications in PVS are organized in the form of parameterize theories comprising definitions, axioms, assumptions, lemmas and theorems.

PVS offers inference rules, proof commands and decision procedures that can be used to prove theorems. The PVS prover is based on *sequent calculus* where each proof goal is a *sequent* consisting of formulas called *antecedents* and *consequents*, such as:

$$\begin{array}{l} \{-1\}\ X_1 \\ \{-2\}\ X_2 \\ \{-3\}\ X_3 \\ \quad .. \end{array}$$

$$\frac{}{\begin{array}{l}\{1\}\ Y_1\\ \{2\}\ Y_2\\ \{3\}\ Y_3\\ \quad ..\end{array}}$$

where formulas (X_i) above the line represent the antecedents and formulas (Y_j) below the line are the consequents. In a sequent, the *conjunction* of the antecedents implies the *disjunction* of the consequents, such as $(X_1 \wedge X_2 \wedge X_3 \ldots) \supset (Y_1 \vee Y_2 \vee Y_3)$. Here, a simple example is presented for the sum function that is defined recursively in PVS.

```
n: VAR nat
 sum(n): RECURSIVE nat = (IF n = 0 THEN 0
    ELSE n + sum(n - 1) ENDIF)  MEASURE n
the: THEOREM sum(n) = (n * (n + 1))/2
```

Theorem *the* in this example is proved by induction. The base case is proved with *expand* "*sum*" and "*propax*" proof commands. Whereas, the inductive case is proved with commands: *skosimp, expand "sum"* + and *assert*. During proof construction, PVS generates a graphical proof tree in which the remaining proof obligations are the tree leaves. If a proof gets stuck, then this tree helps to see where the proof went wrong. More details on PVS can be found in [36].

B. SPM AND HUIM

The data mining field deals with the problem of discovering meaningful hidden information (knowledge) in large datasets with the help of techniques developed by integrating machine learning with statistics. Extracted information is then transformed into a knowledge-base that can be used for decision making and data understanding. Data mining has gained increased attention in recent years due to the availability of large amount of data in different applications belonging to various fields. Some important data mining tasks are: classification, prediction, clustering, pattern mining and anomaly analysis [1]. In data mining, pattern mining techniques are used to find interesting and useful patterns that are hidden in large datasets or databases. Such techniques can be used on datasets of different types that ranges from sequence and transactions to strings and graphs [2].

Traditional pattern mining techniques are used in many applications. However, these techniques are inappropriate for analyzing data with time or that is sequentially ordered. They fail to find patterns that involve temporal relationships between events or symbols. Sequential Pattern Mining (SPM) [15], a special case of structured data mining was used to address this problem. SPM discovers useful and important subsequences in a set of sequences of symbols, where the importance of a subsequence is measured through different parameters such as its occurrence frequency, profit and length. High utility pattern mining, an emerging topic in SPM, consists of discovering patterns having a high importance in databases. Among the various kinds of high utility patterns (HUP) that can be discovered in databases, high utility itemsets (HUIs) are the most studied. An HUI is a set of values that appears in a database and has a high importance to the user, as measured by a utility function. High utility itemset mining (HUIM) generalizes the problem of frequent itemset mining by considering item quantities and weights [11]. SPM and HUIM has been used in the past in many real-life applications such as recommendation systems, text analysis, e-learning, web-page click-stream analysis, bioinformatics, energy reduction in smart-homes, users preferences identification and market-based analysis. More details on SPM and HUIM can be found in surveys [15] and [11], respectively.

III. PROOF PROCESS LEARNING APPROACH

The proposed HUIM-based proof process learning approach for PVS is shown in Figure 1. It consists of two main parts:

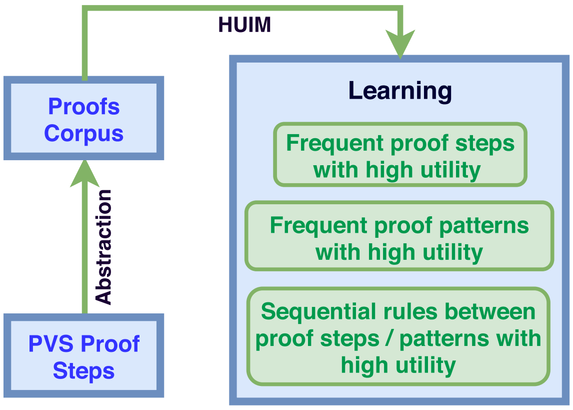

HUIM
Proofs Corpus
Abstraction
PVS Proof Steps
Learning
Frequent proof steps with high utility
Frequent proof patterns with high utility
Sequential rules between proof steps / patterns with high utility

FIGURE 1. HUIM-based proof learning approach.

1) **Proof corpus Development**: Proof steps for theorems/lemmas in PVS theories are converted to a proof corpus. Each complete proof in the corpus is abstracted to a sequence of proof commands/decision procedures and each proof command/decisison procedure is assigned a utility value.
2) **Learning through HUIM**: HUIM algorithms are used on the corpus to discover the common proof steps/patterns and top-k patterns with high utility as well as the sequential relationships between proof steps and patterns.

The PVS data (proofs) is assembled first so that HUIM algorithms can be used. Moreover, the proof corpus should satisfy certain minimum requirements to make it more suitable for learning. For example:

- It is stored in a computational and electronic form.
- It contains many examples of proofs that offer diversity in kinds of proof steps. The corpus should have different proof steps so that useful proof patterns and the dependency of proof steps can be discovered.
- It is transformed in a suitable abstraction, so that no meaningful information from the proofs is left out. To achieve this, we use the "*proof sequences to integers*" abstraction. In such abstraction, each proof command is converted to a distinct positive integer.

The proof development process in PVS is interactive in nature and it follows the *sequent-style* proof representation. A user first provides the property (in the form of a lemma or theorem) that is called a proof goal. User then applies proof commands, inference rules and decision procedures to solve the proof goal. The action resulting from a proof command, inference rule or decision procedure is referred to as a *PVS proof step* (*PPS*) here. A *PPS* may either prove the goal or generates another sequent or divide the main goal into sub-goals. The proof development process for a theorem or lemma is completed when the sequent or all the sub-goals are proved. After proof development, PVS saves the proof scripts of a theory in a separate proof file. These files contain *PPS* with some other information related to PVS. After removing the redundant information from the proof files, the complete proof is a sequence of *PPS*.

Let $PS = \{PPS_1, PPS_2, \ldots, PPS_m\}$ represent the set of *PPS*. A proofs corpus is a set of records (transactions), called proofs, denoted as $PD = \{T_0, T_1, \ldots, T_n\}$, where each proof T_c is a set of proof commands (i.e. $T_c \subseteq PS$), and has a unique identifier c called its TID. For example, consider the proof corpus shown in Table 1. It contains five transactions T_0, T_1, T_2, T_3 and T_4. The record T_2 indicates that three proof commands (*skosimp, expand, propax*) were used in the proof of a theorem/lemma.

TABLE 1. **A sample of a proof corpus.**

TID	Proof Sequence
T_0	⟨{*inst 1 "lambda (x,y: sequence[Time]): false", grind*}⟩
T_1	⟨{*skosimp, expand "Teq", flatten, assert*}⟩
T_2	⟨{*skosimp, expand "Fifon", propax*}⟩
T_3	⟨{*skeep, expand "Tle", typepred "<", expand "strict_order?", flatten, expand "transitive?", inst -2 "T(s1)""T(s2)""T(s3)", assert* }⟩
T_4	⟨{*induct n*},{*expand "sum", propax*},{*skosimp, expand "sum" +, assert*}⟩

The goal of frequent proof steps mining is to discover those proof steps that have high support (occur frequently). A *proof steps set PSS* is a set of proof commands such that $PSS \subseteq PS$. $|PSS|$ denotes the set cardinality. *PSS* has a length k (called *k-PSS*) if it contains k proof commands (i.e., $|PSS| = k$). For example, consider that $PS = \{skolem!, flatten, inst?, split, beta, iff, assert\}$. The set {*skolem!, flatten, assert*} is a *PSS* that contains three proof commands.

The measure that is used mostly in frequent pattern mining is the *support* measure. The support of a *PSS* in a proof corpus *PD* is the total number of transactions (proof sequences) that contain the *PSS*, and is defined as $sup(PSS) = |\{T | PSS \subseteq T \wedge T \in PD\}|$. For example, the support of {*skosimp, expand*} in Table 1 is 3 as it appears in three proof records (T_1, T_2 and T_4).

A *PSS* is a frequent sequence iff $sup(PSS) \geq minsup$, where *minsup* (minimum support) is determined by the user. One main limitation of frequent sequence or pattern mining is that discovered sequences or patterns are not always useful or provide important information.

To overcome this, HUIM can be used, where proof commands are annotated with numerical values and patterns are selected on the basis of a user-defined utility function. The task of HUIM consists of finding patterns in a quantitative database. Some additional information is provided, such as the items quantities in transactions and weights that indicate the relative importance of each item. Two utility functions are generally used.

The first one is the external utility, where each *PPS* is associated with a positive number $p(PPS)$. The positive value of a *PPS* as external utility represents the importance of that *PPS*. Furthermore, every *PPS* that appears in a transaction (proof sequence) T_c has a positive number $q(PPS, T_c)$, called its internal utility. This value represents the quantity (occurrence) of *PPS* in T_c.

These utilities are explained in more details with the proof corpus shown in Table 2. The columns (excluding the first one) represent the number of *PPS* in a particular transaction (proof). In this example, the set of *PPS* is *PS* = {*skosimp, expand, flatten, inst, propax, skeep, typepred, assert, grind, split*}. The corpus in Table 2 contains ten transactions ($T_0, T_1, \ldots T_9$). The transaction T_3 shows that *PPS skosimp, expand* and *assert* were used to prove that transaction, and that have internal utilities of 1, 2, and 1 respectively. Table 3 lists the external utilities of *PSS*. External utilities were assigned to *PPS* on the basis of their importance: the more the *PPS* is used in the corpus, the higher the external utility value for that *PPS*.

Now, the overall utility of a *PPS* in a transaction T_c is denoted as $u(PPS, T_c)$ and defined as $p(PPS) \times q(PPS, T_c)$. The utility of a *PSS* in a transaction T_c is denoted as $u(PSS, T_c)$ and defined as $u(PSS, T_c) = \sum_{PPS \in PSS} u(PPS, T_c)$ if $PSS \subseteq T_c$. Otherwise $u(PSS, T_c) = 0$. Finally the overall utility of a *PSS* in *PD* is denoted as $u(PSS)$ and defined as $u(PSS) = \sum_{T_c \in g(PSS)} u(PSS, T_c)$, where $g(PSS)$ is the set of transactions that contain *PSS*.

For example, the utility of *PPS skosimp* in transaction T_2 is $u(skosimp, T_2) = 1 \times 3 = 3$. The utility of the *PSS* $\{sksoimp, expand\}$ in T_2 is $u(\{skosimp, expand\}, T_2) = u(skosimp, T_2) + u(expand, T_2) = 1 \times 3 + 1 \times 6 = 9$. The utility of *PSS* $\{sksoimp, expand\}$ in *PD* is $u(\{skosimp, expand\}) = u(skosimp) + u(expand) = u(skosimp, T_2) + u(skosimp, T_3) + u(skosimp, T_6) + u(skosimp, T_7) + u(skosimp, T_8) + u(skosimp, T_9) + u(expand, T_2) + u(expand, T_3) + u(expand, T_6) + u(expand, T_7) + u(expand, T_8) + u(expand, T_9) = 3 + 3 + 3 + 3 + 3 + 3 + 6 + 12 + 6 + 36 + 24 + 24 = 126$. Thus, the utility of $\{sksoimp, expand\}$ in the *PD* can be interpreted as the total profit generated by *PPS skosimp* and *expand* when they are used together. The problem of high utility *PSS* mining is defined as follows:

A *PSS* is a high-utility *PSS* if its utility $u(PSS)$ is no less than a user-defined *minutil* threshold (i.e. $u(PSS) \geq minutil$). The problem of high-utility *PSS* mining is to find

TABLE 2. A transactional proof corpus with internal utility values.

TID	skosimp	expand	skolem!	flatten	inst	inst?	propax	skeep	typepred	assert	grind	split
T_0	0	0	0	0	1	0	0	0	0	0	1	0
T_1	0	4	0	1	1	0	1	1	1	1	0	1
T_2	1	1	0	0	0	0	0	0	0	1	0	0
T_3	1	2	0	0	0	0	0	0	0	1	0	0
T_4	0	3	0	1	1	0	0	1	1	1	0	0
T_5	0	3	0	1	1	0	0	1	1	1	0	0
T_6	1	1	0	0	0	0	0	0	0	1	0	0
T_7	1	6	0	1	0	0	1	0	0	2	0	1
T_8	1	4	0	1	0	0	1	0	0	2	0	1
T_9	1	4	0	1	0	0	0	0	0	5	0	1

TABLE 3. *PPS* external utility values.

Item	profit	Item	profit
skosimp	3	*expand*	6
skolem!	1	*flatten*	2
inst	3	*inst?*	2
propax	1	*skeep*	1
typepred	1	*assert*	5
grind	4	*split*	2
induct	2	*skosimp**	2
lemma	1	*use*	1
hide	1	*hide-all-but*	1
case-replace	1	*prop*	1
rewrite	1		

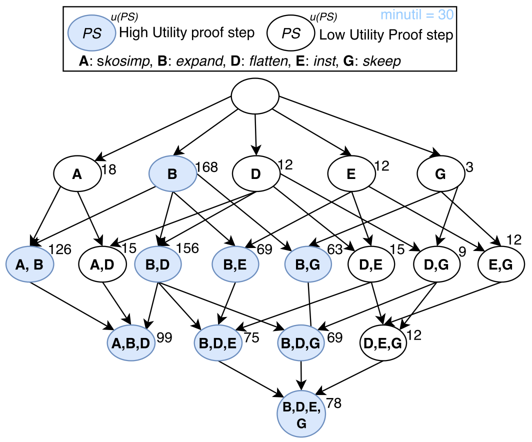


FIGURE 2. The search space of high utility proof step mining.

all high-utility proof *PSS*, given a *minutil* threshold that is set by the user.

A. CHALLENGES

The goal in high utility *PSS* mining is to enumerate all the patterns for *PSS* in the corpus with a utility greater than or equal to the *minutil*. However, it is difficult to achieve the goal because of three reasons.

First, the number of *PPS* can be very large. Generally, if a corpus contains m distinct *PPS*, then there are $2^m - 1$ possible *PSS* (not including the empty set). For example, if *PS*= {*skosimp*, *expand*, *assert*}, the possible *PSS* are {*skosimp*}, {*expand*}, {*assert*}, {*skosimp*, *expand*}, {*skosmp*, *assert*}, {*expand*, *assert*}, and {*skosimp*, *expand*, *assert*}. Thus, there are $2^3 - 1 = 7$ *PSS* for this particular *PS*. In the naive approach, the utilities of all possible *PSS* are counted by scanning the corpus and then only *PSS* having high utility values are kept. This naive approach is inefficient despite the fact that it produces the correct result. The reason is that the number of possible *PSS* can be very large. For example, if the corpus has 20 *PPS*, we need to calculate the utilities of $2^{20} - 1$ possible *PSS*. This is very hard to achieve with the naive approach as the size of the search space (that indicates the total number of possible *PSS*) is very large even if there are few proofs in the corpus. In fact, the overall size of the search space also depends on how similar the proofs are in the corpus, how large the utility values are, and on whether the *minutil* threshold is low or high.

The second reason is that high utility *PSS* are often scattered in the search space. Thus, many *PSS* must be considered by an algorithm before it can find the actual high utility *PSS*. To illustrate this, Figure 2 provides the Hasse graph of the search space for some *PSS* from Table 2. In a Hasse graph, each node represents the possible *PSS* and a uni-directional edge is drawn from one PSS_1 to another PSS_2 if and only if $PSS_1 \subseteq PSS_2$ and $|PSS_1| + 1 = |PSS_2|$. In Figure 2, high utility *PSS* are shown with light blue nodes, while low utility *PSS* are represented using white nodes. The utility value of each *PSS* is also indicated along the node. It can be observed from Figure 2 that the utility of a *PSS* can be higher, lower or equal to the utility of any of its supersets/subsets. For example, the utility of the *PSS* $\{D, E\}$ is 15, while the utility of its supersets $\{D, E, G\}$ and $\{B, D, E, G\}$ are 12 and 78, respectively. This means that the utility measure is neither monotonic nor anti-monotonic. Because of this property, the high utility *PSS* appear scattered in the search space. Thus, the problem of high utility *PSS* mining is more difficult than the problem of frequent *PSS*, because the support measure is monotonic in the later case. This means that the support of a *PSS* can be either higher or equal to the frequency of any of its supersets.

The third reason is due to the interactive proof process in PVS. During the proof development process, users are required to formalize their inputs with (1) *PPS*, and (2) arguments for those *PPS*. For example the *PPS* (*expand "Teq"*) expands the function *Teq* and *typepred* < gives the type predicate of <. Therefore, a proof goal in PVS can be considered as a *context-PPS* pair, where *context* contains the information about the current hypotheses, variables and the goal that needs proving. The goal may contains a set of sub-goals. The user is required to guide the proof process towards completion by suggesting which *PPS* (and arguments) to use. However, the arguments depends on the specification (particularity on variables and functions declarations) inside the theory and on the proof goal. This means that arguments for a particular *PPS* can be different for different theories and different proof goals. To avoid this, we focus on *PPS* only in this work. We believe that adding arguments information would restrict the learning model to work well for only one (or some related) theory.

Because the utility measure is neither monotonic nor anti-monotonic, strategies for reducing the search space used in frequent pattern mining cannot be used directly in the problem of high utility pattern mining. Still, some fast algorithms are designed in recent years that can avoid the scanning of all possible itemsets in the search space and can find all high utility patterns. Some of these algorithms that we used in this work to mine PVS proofs are described in the next section.

IV. ALGORITHMS FOR HUIM

For HUIM, various algorithms can be found in the literature. All of these algorithms generally have the same input and output. The differences lie in the strategies and data structures that are used in these algorithms for searching HUIs. More specifically, they differ in (1) whether a depth-first or breadth-first search is used, (2) the type of database representation, (3) how the next itemsets, that needs further exploration in the search space, is determined, and (4) how the utility of itemsets is computed to check whether they satisfy the minimum utility constraint. We discuss some important HUIM algorithms next.

A. TWO-PHASE ALGORITHMS

The first algorithms to find HUIs perform two phases, that is why they are known as two-phase algorithms. The three famous two-phase algorithms are Two-Phase algorithm [30], IHUP [3] and UP-Growth [42]. The basic idea in these algorithms is to use a monotonic measure (known as TWU (Transaction Weighted Utilization)) that puts an upper-bound on the utility measure. The TWU is defined as follows:

In the context of this paper, the TWU of a *PSS* is defined as the sum of the utilities of transactions containing *PSS*, i.e. $TWU(PSS) = \sum_{T_c \in g(PSS)} TU(T_c)$, where the transaction utility (TU) of a transaction T_c represents the sum of the utilities of all the *PPS* in T_c, i.e. $TU(T_c) = \sum_{PPS \in T_c} u(PPS, T_c)$.

The transaction utilities of $T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8$ and T_9 are respectively 7, 39, 14, 20, 30, 30, 14, 54, 42, 56. The TWU of single *PPS* *skosimp*, *expand*, *flatten*, *inst*, *propax*, *skeep*, *typepred*, *assert*, *grind*, *split* are respectively 200, 309, 221, 106, 135, 99, 99, 299, 7 and 191. The TWU of the *PSS* $\{flatten, inst\}$ is $TWU(\{flatten, inst\}) = TU(T_1) + TU(T_4) + TU(T_5) = 39 + 30 + 30 = 99$. The TWU measure is an upper-bound on the utility measure and can be used to reduce the search space.

For example, the utility of the *PSS* $\{expand, flatten, inst, propax\}$ is 30, and $TWU(\{expand, flatten, inst, propax\}) = 39$. Thus, any supersets of $\{expand, flatten, inst, propax\}$ cannot have a TWU and a utility greater than 39. So, if the *minutil* is set to a value greater than 39, all supersets of $\{expand, flatten, inst, propax\}$ can be eliminated from the search space, because their utilities cannot be greater than 39. Algorithms such as Two-Phase algorithm [30], IHUP [3] and UP-Growth [42] use this property to prune the search space.

In two phase algorithm, high utility itemset (*PSS* in our case) are mined in two phases. In the first phase, a set of candidates is found. In the second phase, the utility of the found candidates is computed by scanning the database. Itemsets with low utility are filtered and the high utility itemsets are returned [27]. However, this approach is inefficient as the set of candidate itemsets found with phase 1 can be very large. Running the second phase to evaluate these candidates can be very costly. In the worst case, all candidate itemsets are compared to all transactions of the database. Therefore, the performance of two phase algorithms highly depends on the number of generated candidates in the first phase. This problem is addressed by designing one-phase algorithms, which are described next.

B. ONE-PHASE ALGORITHMS

One-phase based HUIM algorithms do not generate candidates. Such algorithms directly calculate the utility of each considered pattern in the search space. The advantage of this approach is that an itemset can be identified immediately as a low utility or high utility itemset. Moreover, candidates are not stored in the memory. The concept of one-phase algorithm was first presented in [29], which was later improved and more efficient one-phase algorithms have been designed such as FHM. These algorithms also introduced the notion of novel upper-bounds on the utility of itemsets that are based on the exact utility of each itemset, and can thus prune a larger part of the search space compared to the TWU measure. The most popular type of HUIM algorithms are based on the utility-list structure.

FHM (Fast High-utility itemset Mining) is a one-phase algorithm that employs a depth-first search to explore the search space. During the searching process, the FHM algorithm builds a *utility-list* for each visited itemset. The utility-list stores information about the utility of the itemset in the transactions where it appears as well as the information related to the utilities of remaining items in these transactions. Utility-lists allow the quick utility computation of an itemset and upper-bounds on the utility of its super-sets. This is done without scanning the whole database. Furthermore, utility-lists of *k-itemsets* ($k > 1$) can be built quickly by joining utility-lists of shorter patterns. The utility-list is defined as follows:

Let X represents an itemset and D represents a quantitative database. Assume that a total order $\prec$ is defined on the set of items I that appears in D. The utility-list $ul(X)$ in D is a set of tuples $(tid, iutil, rutil)$ for each transaction T_{tid}

that contains X. The *iutil* shows the utility of X in T_{tid}, i.e. $u(X, T_{tid})$. Whereas, the *rutil* element is defined as $\sum_{i \in T_{tid} \wedge i > x \forall x \in X}(i, T_{tid})$. The utility-list of an itemset uses the remaining utility upper-bound to prune the search space. The utility-list structure used by the FHM algorithm is said to be a vertical database representation. More details on FHM can be found in [16]. The optimized version of FHM is FHM+ [13] that efficiently discover HUIs by putting upper-bounds on the utility of itemsets by using length constraints with a technique known as *Length Upper-bound Reduction* (LUR).

Utility-list based algorithms are easy to implement and they are efficient. However, these algorithms have some important limitations. First, they may search for some itemsets that are not present in the database, as the itemsets are generated by combining itemsets, without scanning the database. Thus, they may waste a lot of time in building the utility-lists of non-exisiting itemsets. Second, these algorithms sometimes uses a lot of memory in building a utility-list for each visited itemset in the search space.

One-phase algorithms based on pattern-growth solve several drawbacks of utility-list based algorithms. These algorithms explore the search space by scanning the database. Thus, itemsets that exist in the database are considered only. The D2HUP [28] is the first such algorithm. D2HUP algorithm performs a depth-first search and uses a hyper structure to represents the database and its corresponding projections. However, creating and updating the hyperstructure can be quite costly. Another algorithm is the EFIM algorithm [46] that is inspired by the LCM algorithm. EFIM processes each itemset in the search space in linear time and space. It performs a depth-first search using an horizontal database representation. Moreover, two novel upper-bound (the *local-utility* and *subtree-utility*) are used in EFIM to reduce the search space. Moreover, another novel array-based utility counting technique (named *Fast Utility Counting*) is used to compute the two upper-bounds in linear time and space. In last, EFIM integrates *high-utility database projection* (HDP) and *high-utility transaction merging* (HTM) to reduce the cost of database scans.

C. TOP-K HUIM ALGORITHMS

In traditional HUIM algorithms, the value *minutil* threshold greatly affects the execution time, memory use and the number of generated patterns. An algorithm may generate a huge number of patterns if *minutil* is set too low. This may make the algorithm to run slow and use much memory. Similarly, the algorithm may generate a few or no patterns if *minutil* is set too high. For these limitations, the problem of *top-k* HUIM was proposed [41]. Such algorithms discover the *k itemsets* with the highest utility by replacing the *minutil* parameter with k. The working of a *top-k* HUIM algorithm is as follows. Initially, it starts exploring the search space by setting an internal *minutil* to 0. Then, after finding the k high utility itemsets, *minutil* is increased to the pattern utility that has the lowest utility among the current *top-k* patterns. The search goes on for each found high utility itemset and the set of the current *top-k pattern* is updated alongside the internal *minutil*. The set of the *top-k* high utility itemsets is returned after the algorithm terminates. The problem of *top-k* HUIM is more difficult than the problem of HUIM as the former start with internal *minutil* $= 0$. In the past, various *top-K* HUIM algorithms have been proposed. In this paper, two algorithms, TKU (*Top-K* utility itemsets) and TKO (*Top-K* utility itemsets in one phase), are used. The details for both algorithms can be found in [41]. A comparison of the considered HUIM algorithms for various parameters is provided in Table 4.

TABLE 4. Comparison of algorithms used for mining PVS proofs.

Algorithm	ST	Phase	DBR	Based on
Two-Phase [30]	BF	2	H	Apriori [1]
FHM [16]	DS	1	V	Eclat [45]
FHM+ [13]	DS	1	V	FHM [16]
D2HUP [28]	DS	1	V	H-Mine [37]
EFIM [46]	DS	1	V	LCM [43]
TKU [41]	DS	2	H	UP-Growth [42]
TKO [41]	DS	1	V	HUI-Miner [28]

ST: Search type: DBR: Database representation, BF: Breadth-first, DF: Depth-first, H: Horizental, V: Vertical.

D. EVOLUTIONARY-BASED AND SEQUENTIAL RULE FINDING ALGORITHMS

Besides traditional approaches, evolutionary techniques are also used to mine high-utility patterns. Evolutionary algorithms are optimization and search technique that can solve complex and often highly nonlinear problems. They can investigate very large problem spaces to find the best solution based on fitness functions under a set of multiple constraints. HUIM approaches that are based on genetic algorithm (called HUIF-GA) and on particle swarm optimization called HUIF-PSO) were proposed in [40].

Frequent or high utility patterns with low confidence usually play no important role in decision making or prediction. To address this limitation, it was proposed to discover sequential rules by not only considering their support but also their confidence [12]. In the context of this work, a sequential rule $X \rightarrow Y$ is a relationship between two *PSS*s $X, Y \subseteq PS$, such that $X \cap Y = \emptyset$ and $X, Y \neq \emptyset$. The rule $r : X \rightarrow Y$ means that if items of X occur in a sequence, items of Y will occur afterward in the same sequence. X is contained in S_α (written as $X \sqsubseteq S_\alpha$) iff $X \subseteq \bigcup_{i=1}^{n} \alpha_i$. A rule $r : X \rightarrow Y$ is contained in S_α ($r \sqsubseteq S_\alpha$) iff there exists an integer k such that $1 \leq k < n$, $X \subseteq \bigcup_{i=1}^{k} \alpha_i$ and $Y \subseteq \cup_{i=k+1}^{n} \alpha_i$. Furthermore, let $seq(r)$ and $ant(r)$ respectively denotes the set of sequences containing r and the set of sequences containing its *antecedent*, i.e. $seq(r) = \{S_\alpha | S_\alpha \in PD \wedge r \sqsubseteq S_\alpha\}$ and $ant(r) = \{S_\alpha | S_\alpha \in PD \wedge X \sqsubseteq S_\alpha\}$. The confidence of r in PD is defined as $conf_{PD}(r) = |seq(r)|/|ant(r)|$ and the support of r in PD is defined as $sup_{PD}(r) = |seq(r)|/|PD|$.

A rule r is a *frequent sequential rule* iff $sup_{PD}(r) \geq minsup$ and r is a *valid sequential rule* iff it is frequent and $conf_{PD}(r) \geq minconf$, where the thresholds *minsup*, *minconf* $\in [0, 1]$ are set by the user. Mining sequential rules in a corpus deals with finding all the valid sequential rules. ERMiner [12] is the state-of-the-art algorithm to mine sequential rules in a sequence dataset. It relies on a vertical database representation and represents the search space of rules using equivalence classes of rules having the same antecedent or consequent. It employs two operations (left and right merges) to explore the search space of frequent sequential rules, where the search space is pruned with the *sparse count matrix* (SCM) technique, which makes ERMiner more efficient than other sequential rule finding algorithms.

The utility of a rule r in a sequence s_c is defined as $u(r, s_c) = \sum_{i \in X \cup Y} u(i, c_s)$ iff $r \subseteq c_s$. Otherwise it is 0. The utility of r in a PD is defined as $u_{PD}(r) = \sum_{s \in PD} u(r, s)$. A rule r is a high-utility sequential rule iff $u_{PD}(r) \geq minutil$ and r is a valid rule. The problem of mining high utility sequential rules from a sequence database is to discover all high utility sequential rules. For mining high utility sequential rules, HUSRM (High Utility Sequential Rule Miner) is used. It explores the search space of sequential rules using a depth-first search. To prune the search space of sequential rules, the HUSRM algorithm adapts the concept of sequence estimated utility. HUSRM first scans the database to build all sequential rules of size 1×1. Then, it recursively performs left/right expansions starting from those sequential rules to generate larger sequential rules. More details on HUSRM can be found in [47].

V. EXPERIMENTS AND RESULTS

All the experiments are performed on an HP laptop with a fifth generation Core i5 processor and 8 GB RAM. For the case study, we select our previous works [32], [33], where PVS is used for the analysis and verification of Reo connectors composed of untimed, timed and probabilistic channels. The main reason to select the proofs in [32], [33] is that we are extending the formalization framework to cover the stochastic [5] and hybrid [8] behavior of Reo connectors. We believe that this proof learning approach will not only enable us to comprehend the proof process for stochastic and hybrid connectors but also can be considered far more effective in providing the necessary guidance to attain the proofs of such connectors.

SPMF data mining library is used to analyze the *PD*. It is an open-source and cross-platform framework that is developed in Java and offers implementations for more than 190 data mining algorithms, including several high utility pattern mining algorithms. More detail on SPMF can be found in [14].

A. CASE STUDY

Reo [4] is a channel-based exogenous coordination language that allows the construction of complex *connectors* from primitive *channels* through compositional operators. Connectors in Reo provide the protocol for controlling and organizing the communication, synchronization and cooperation between components. Each channel in Reo has two channel end types *source* or *sink*. A source channel end accepts data into the channel and a sink channel end dispenses data out of the channel. Few primitive channel types in Reo are shown in Figure 3.

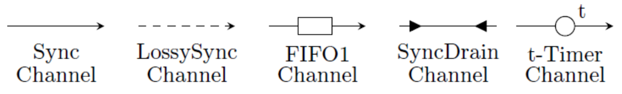


FIGURE 3. Some primitive channels in Reo.

For example, the *synchronous (Sync) channel* has one source and one sink end. Input/Output (I/O) operations can succeed only if the writing operation at source end is synchronized with the read operation at its sink end. Similarly, a *lossy synchronous (LossySync) channel* is a variant of synchronous channel that accepts all data through its source end. The written data is lost immediately if no corresponding read operation is available at its sink end. A *FIFO1 channel* channel has one buffer cell, one source end and one sink end. The channel accepts a data item whenever the buffer is empty. The data item is kept in the buffer and dispensed to the sink end later in the FIFO order.

Figure 4 shows a connector composed from combining two channels (*Sync* channel (*AB*) and *FIFO1* channel (*BC*)). This connector can accepts data items at source node *A* and stores the data items in the buffer, before dispensing them through the sink node *C*. The mixed node *B* allows the data items to move from one channel to anohter channel without any change.

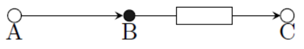


FIGURE 4. A connector composed of a Sync and a FIFO1 channel.

The untimed and timed connectors behavior in PVS [32] was formalized by means of data-flows on its sink and source nodes. In PVS, record structure named *TD* is used to represent the *timed-data* sequences on sink and source nodes, where *time* is defined as a *positive real number* ($\mathbb{R}^+$) and *data* is defined as a *positive* type.

```
Time: Type = posreal
Data: Type+
TD = TYPE = [# T: sequence[Time],
               D: sequence[Data] # ]
Input, Output: VAR TD
```

Input and *Output* are declared as variables of type *TD*. Some predicates are used for untimed channels specification. For example, *Teq* takes two *TD* sequences and returns *true* if the time of two sequences are equal. *Tle* represents that time of the first sequence is strictly less than the second sequence and *Deq* shows the equality of data: data sequence at *Input* is equal to data sequence at *Output*. Similarly for timed channels, some more predicates are specified. These predicates are similar to the untimed predicates with one of the time sequences is added by a *t* time delay.

```
Teq(Input,Output):bool = T(Input) = T(Output)
Tle(Input,Output):bool = T(Input) < T(Output)
Deq(Input,Output):bool = D(Input) = D(Output)
```

With *TD* and predicates, Reo channels such as *Sync* and *FIFO1* are specified as follows:

```
Sync(Input,Output):bool = Teq(Input,Output) &
                           Deq(Input,Output)
Fifo1(Input,Output):bool= Tle(Input,Output) &
  Tle(Output,next(Input)) & Deq(Input,Output)
```

The probabilistic connectors behavior in [33] is formalized with *timed data distribution* (*TDD*), where *T* is a sequence of time points, and *DD* is a sequence of *data distributions*. For untimed/timed channels, data is defined as a *positive type*. To capture the probabilistic behavior, data is defined as a function of type $[T \rightarrow real]$ (where *T* is a positive (non-empty) type).

```
Time: Type = posreal
Data: TYPE [T -> real]
DD: TYPE = [Data, df]
TDD: TYPE = [# T: sequence[Time],
               D: sequence[DD] #]
Input, Output: VAR TDD
```

where *df* is the distribution function for a real-valued random variable *X*.

Three main composition operators (flow through, replicate and merge) are used in Reo for connector construction. Flow through and replicate operators can be achieved explicitly in PVS, whereas merge operator is defined inductively. The complete details on the specification for Reo channels, operators and predicates can be found in [32], [33], [38].

Primitive channels in Reo can be combined together to form connectors. The connector in Figure 4 can be specified in PVS by combining the *Sync* and *FIFO1* channels, such as *Sync(A, B)* & *FIFO1(B, C)*. Let *a*, *b*, *c* denote the time sequences when the corresponding data sequence flows through nodes *A*, *B* and *C* respectively. According to the semantics of *Sync* and *FIFO1* channels, $a = b < c$. Let α, γ represent the data sequences being observed at nodes *A* and *C* respectively, and $\alpha = \gamma$. In PVS, these results are proved with the following theorem.

Theorem 1: $Sync(A, B) \wedge Fifo1(B, C) \Rightarrow Tle(A, C) \wedge Teq(A, B) \wedge Deq(A, C)$

The proof steps for theorem 1 are shown in Figure 5.

The proof corpus *PD* contains the complete proofs that are used to prove important properties of connectors as well as the proofs for theories, such as probability theory, that are used in the modeling of Reo connectors.

B. RESULTS

Results obtained by applying the algorithms discussed in Section IV on the proof corpus are discussed in this section.

We first run Two-Phase algorithm on the *PD*. The *minutil* threshold was set to 60. The total candidates (high utility *PSS*) searched by the algorithm were 721, out of which 209 were selected by the algorithm as high utility *PSS*. Some obtained results with high utility (Util.) and support (Sup.) are listed in Table 5. Note that the support for each *PSS* was calculated by the algorithm. The first pattern shows that the {*expand, flatten*} was found in 48% of proof sequences in *PD* with

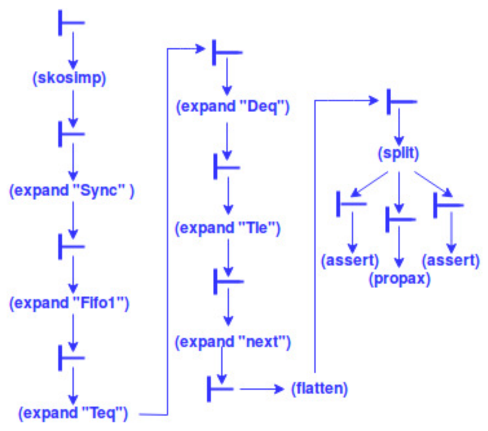


FIGURE 5. PVS proof tree for theorem 1.

TABLE 5. Extracted high utility proof patterns with Two-Phase algorithm.

pattern	Util.	Sup.
expand, flatten	60	0.48%
skosimp, expand, grind	63	0.11%
skosimp, expand, skolem!, flatten	111	0.111%
skosimp, expand, inst?, grind	115	0.047%
expand, flatten, inst?, typepred, assert, grind	127	0.074%
skosimp, expand, flatten, inst, assert, split, induct	111	0.037%
skosimp, expand, flatten, inst?, typepred, assert, grind	107	0.074%
skosimp, expand, flatten, inst, typepred, assert, induct, skosimp, hide-all-but*	90	0.037%
expand, flatten, ist, induct, skosimp, hide-all-but*	130	0.37%
skosimp, expand, inst, typepred, induct, skosimp, hide-all-but*	189	0.37%
expand, flatten, typepred, induct, skosimp, hide-all-but*	119	0.037%
flatten, inst, typepred, assert	79	0.1%

utility of 60. Most of the generated patterns generally started from the *skosimp*, and *expand PPS*. Table 5 provides some useful information (*PSS*) that are used in the verification of Reo channels and connectors. It is important to point out here that some discovered patterns (such as patterns 4 and 6) in Table 5 have very low support, but their utility value is high. Such patterns would not be discovered with traditional frequent patterns approaches in [34].

The comparison of Two-Phase (TP) algorithm with various one phase algorithms is presented in Table 6. All algorithms listed in Table 6 except FHM+ has one parameter. The results listed in Table 6 is for the following parameters: *minutil* = 10, *minim pattern length* = 2 and *maximum pattern length* = 12. The performance of D2HUP was the worst in all of them in terms of parameters listed in Table 6. Whereas, the performance of all other algorithms were almost similar with negligible difference.

Discovered proof sequences with fewer proof steps are generally more important than long proof sequences. The reason is that long *PSS* represent those proof sequences that are more specific, and thus occur rarely. FHM+ can be used to find proof sequences that are more useful to users while filtering those that may be less useful. Some short proof

TABLE 6. Algorithms comparison.

Models	TP	FHM	FHM+	D2HUP	EFIM
Time (ms)	18	17	152	**2600**	19
Memory (MB)	–	59	67	**105**	58
HUIC	738	729	809	**30126**	730
CC	931	927	929	**3188**	929

HUIC: high utility itemset count, CC: candidate count.

TABLE 7. Extracted patterns with FHM+.

pattern	Util.
skosimp, expand	261
skosimp, flatten	78
skosimp, assert	160
expand, flatten	776
expand, assert	1089
skosimp, expand, flatten	162
expand, flatten, assert	1226
expand, assert, split	481
expand, expand, flatten, assert	232
skosimp, expand, flatten, assert, split	116

sequences obtained with FHM+ are listed in Table 7. As we are interested in short proof sequences, the *minimum length* and *maximum length* were set to 2 and 4, respectively. The total proof sequences searched by FHM+ were 222, out of which 163 were selected as high utility proof sequences. It is important to point out here that the best parameter values for aforementioned algorithms were determined empirically after testing to make sure that we obtain few important patterns and not too many redundant patterns. The utility value of *PSS* in Table 7 represent their overall importance in the proofs of theorems/lemmas. We get some interesting results, like most of the proofs for theorems or lemmas start with the *skosimp*, followed by *expand* and *flatten*. Similarly, we also obtained small proof patterns (with high utility values) that were used frequently at the end of proofs in *PD*, such as {*assert, grind*} and {*typepred, assert, grind*}. Table 8 lists the counts for the total proof sequences searched and generated by FHM+ for two varying parameters.

TABLE 8. Results with FHM+.

Min. Length	Max. Length	PSC	HUPSSC
1	2	38	30
2	3	114	85
2	4	222	163
2	5	469	335
2	6	668	495
2	7	722	539
2	8	735	549

PSC: Proof sequences count, HUPSSC: High utility PSS count

In Table 9, results obtained with *top-k* high utility mining algorithms TKU and TKO are listed. It is important to note here that TKU uses two-phases to discover *top k* HUIs, whereas, TKO uses only one phase. In our case, both algorithms generated the same results. For example for $k = 30$, both algorithms generated the same number of *PSS* with almost same execution time and memory used. The results

TABLE 9. Extracted proof steps/patterns with TKO and TKU.

pattern	Util.
flatten, inst, propax, skeep, typepred, assert, split	15
expand, flatten, inst, propax, skeep, typepred, assert, split	84
expand, expand, flatten, inst, propax, skeep, typepred, assert, split	162
expand, expand, expand, flatten, inst, propax, skeep, typepred, assert, split	132
inst, skeep, typepred	10
inst, skeep, typepred, assert	20
expand, inst, skeep, typepred, assert	112
expand, expand, inst, skeep, typepred, assert	198
flatten, inst, typepred, induct, skosimp, hide-all-but*	33
skosimp, flatten, inst, typepred, induct, skosimp, hide-all-but*	84
skosimp, skosimp, flatten, inst, typepred, assert, induct, skosimp, hide-all-but*	396

obtained with top-k HUIM algorithms were different from the two-phase and one phase based HUIMs algorithms. These algorithms generated the top patterns with high utility in the proofs. For example, the first top pattern in Table 9 shows that *PSS* {*flatten, inst, propax, skeep, typepred, assert, split*} with high utility 15 occurred frequently in the *PD*. Moreover, the utility of a discovered pattern generally increases by adding more *PPS* in the pattern. However, in some cases, such as pattern four in Table 9, the utility of a discovered pattern decreases when a new *PPS expand* is added.

Results obtained by running evolutionary based HUIM algorithms is presented in Table 10. Both algorithms generated the same results. However, HUI-GA was approximately twice slower than PSO, while the memory usage was the same for both of them. The possible reason for this is that the GA-based algorithm use two operators (crossover and mutation) to evolve and discover HUIs. Whereas, PSO uses internal velocity that is used to update the positions of particles (itemsets) towards the optimal value. Moreover, PSO converges faster than GA. One important thing to note here is that patterns obtained by evolutionary-based HUIM are different from those found by traditional HUIM algorithms.

TABLE 10. Frquent proof steps/patterns extracted with HUIF-GA and HUIF-PSO.

pattern	Util.
skosimp, assert	128
flatten, assert	198
skosimp, expand, flatten, assert	173
expand, flatten, assert	166
expand, flatten, inst, assert	61
skosimp, expand, typepred, assert	75
skosimp, expand, flatten, split	65
skosimp, expand, flatten, assert, grind	60
skosimp, expand, typepred, assert, grind	57

Table 11 lists some of the relationships between proof steps/patterns that are discovered through sequential rule mining with the HUSRM algorithm. The confidence (*misconf*) threshold is set to 40%, which means that rules have a confidence of at least 40% (a rule $X ==> Y$ has a confidence of 40% if the set of proof commands in X is

TABLE 11. Sequential rules with high utility in the corpus.

rule	S	C	Util.
skosimp ==> expand	12	1	108
skosimp ==> expand, flatten	7	0.58	77
skosimp ==> expand, flatten, assert	7	0.58	112
skosimp ==> expand, assert	11	0.91	154
induct ==> inst, assert	4	0.43	20
skosimp, expand ==> flatten, split	4	0.43	52
skosimp, expand, assert ==> flatten	4	0.46	64
skosimp, expand, flatten ==> assert	5	0.45	48
skosimp, flatten, skeep, typepred ==> inst, assert	3	1	34
skosimp, expand ==> flatten, assert, split	4	0.43	48

S: Support, C: Confidence

followed by the set of proof commands in Y at least 40% of the times when X appears in a proof sequence). The second column in Table 11 is for the support and the third column indicates the confidence (probability). The fourth column shows the utility value for the rule. For example, the second rule in Table 11 indicates that 58% of the time, the $\{expand, flatten\}$ is followed after the *skosimp* command. With HUSRM algorithm, we found some interesting relationships and dependencies between proof steps and patterns. These results were different from the sequential rules discovered in the frequent proof steps/patterns with no utility values. For comparison, the sequential rules discovered with ERMiner in [34] are shown Figure 6. ERminer can be used to find sequential rule in a sequence database that does not have utility information. The value above the arrow in Figure 6 represents the support for the rule and the value below the arrow represents the confidence.

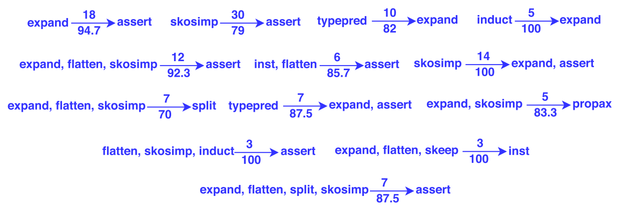

expand 18 94.7 assert
skosimp 30 79 assert
typepred 10 82 expand
induct 5 100 expand
expand, flatten, skosimp 12 92.3 assert
inst, flatten 6 85.7 assert
skosimp 14 100 expand, assert
expand, flatten, skosimp 7 70 split
typepred 7 87.5 expand, assert
expand, skosimp 5 83.3 propax
flatten, skosimp, induct 3 100 assert
expand, flatten, skeep 3 100 inst
expand, flatten, split, skosimp 7 87.5 assert

FIGURE 6. Sequential rules discovered with ERMiner.

In [9], common proof patterns are found in the Isabelle proofs with a variable length Markov Chain. Proofs are represented in a tree structure format, which are linearized, such as the proofs are split into separate sequences and given weights accordingly. However, linearization means losing any important connections (information) between different branches in the proofs due to which interesting patterns may well be lost. Another work [26] used a hybrid method that combines statistical data mining and theory exploration to analyse and automate Coq proofs. Coq proof terms and object types are also represented as a tree. Their aim was to find useful lemmas or hypotheses as arguments of a tactic. However, no such promising terms that can be used as arguments of a tactic were identified. In this work, the proof corpus contains all the necessary important information for pattern discovery and SPM, HUI algorithms, which are more user-friendly and work efficiently on the corpus.

Overall, it was observed through various experiments that SPM and HUIM techniques can be used effectively to learn the proof development process in PVS. Besides HOL4, learning approaches can also be used in other proof assistants such as HOL4 and Coq. Results obtained so far indicate that the total number of proof steps in each proof (abstraction simplicity) and the utility value assigned to each proof command have a direct correlation on the efficiency of HUIM algorithms. These preliminary results indicate that the research direction of linking and integrating evolutionary algorithms with proof assistants is worth pursuing. This approach may have a considerable impact to advance and accumulate human knowledge, especially in the fields of formal logic and computation. The ultimate goal is to develop proof tactics/strategies with useful patterns that can be invoked directly by the user in the proof development process.

VI. CONCLUSION

The proof development process in ITPs is interactive in nature, where users guide the proof searching and are forced to do lots of repetitive work. This makes the proving process in ITPs a cumbersome and a time consuming task. To make the proof process simpler and for providing proof guidance, HUIM-based learning approach is adopted in this work to find the frequent proof steps/patterns and their relationship in PVS theories. Some interesting proof patterns are found with HUIM and obtained results show that the number of proof steps in each proof and the utility value assigned to each proof command have a direct correlation on the efficiency of HUIM algorithms.

In the future, this work can be extended in several directions. First, we will use the SPM and HUIM-based learning approach on the corpora of proof steps for theories included in PVS library, which contains thousands of theorems. This will enable us to develop a more general learning approach for the proofs of new conjunctures (unseen theorems). Another direction is to use evolutionary and heuristics techniques such as genetic programming and particle swarm optimization for the development of PVS strategies and tactic from frequently occurring proof patterns. Some preliminary work is done in [35], where GA is used for proof searching in HOL4. This also shows that the proposed learning approaches are not limited to PVS only, and can easily used with other proof assistants. Last but not the least, it would be interesting to use HUIM and SPM algorithms on the HOL Light [23] and Coq [24] datasets.

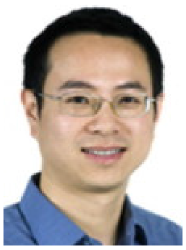
REFERENCES

[1] C. C. Aggarwal, *Data Mining—The Textbook*. Springer, 2015.
[2] C. C. Aggarwal and J. Han, *Frequent Pattern Mining*. Springer, 2014.
[3] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong, and Y.-K. Lee, "Efficient tree structures for high utility pattern mining in incremental databases," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 12, pp. 1708–1721, Dec. 2009.
[4] F. Arbab, "Reo: A channel-based coordination model for component composition," *Math. Struct. Comput. Sci.*, vol. 14, no. 3, pp. 329–366, Jun. 2004.

[5] C. Baier and V. Wolf, "Stochastic reasoning about channel-based component connectors," in *Proc. Int. Conf. Coordination Models Lang.*, in Lecture Notes in Computer Science, vol. 4038. Springer, 2006, pp. 1–15.
[6] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. New York, NY, USA: Springer-Verlag, 2003.
[7] J. C. Blanchette, M. P. L. Haslbeck, D. Matichuk, and T. Nipkow, "Mining the archive of formal proofs," in *Proc. Int. Conf. Intell. Comput. Math.*, in Lecture Notes in Computer Science, vol. 9150. Springer, 2015, pp. 3–17.
[8] X. Chen, J. Sun, and M. Sun, "A hybrid model of connectors in cyber-physical systems," in *Proc. Int. Conf. Formal Methods Softw. Eng.*, in Lecture Notes in Computer Science, vol. 8829. Springer, 2014, pp. 59–74.
[9] H. Duncan, "The use of data-mining for the automatic formation of tactics," Ph.D. dissertation, School Inform., Univ. Edinburgh, Edinburgh, U.K., 2007.
[10] M. Färber and C. E. Brown, "Internal guidance for Satallax," in *Proc. Int. Joint Conf. Automated Reasoning*, in Lecture Notes in Computer Science, vol. 9706. Springer, 2016, pp. 349–361.
[11] P. Fournier-Viger, J. Chun-Wei Lin, T. Truong-Chi, and R. Nkambou, *A Survey of High Utility Itemset Mining*. Springer, 2019, pp. 1–45.
[12] P. Fournier-Viger, T. Gueniche, S. Zida, and V. S. Tseng, "ERMiner: Sequential rule mining using equivalence classes," in *Proc. Int. Symp. Intell. Data Anal.*, in Lecture Notes in Computer Science, vol. 8819. Springer, 2014, pp. 108–119.
[13] P. Fournier-Viger, J. C. Lin, Q. Duong, and T. Dam, "FHM+: Faster high-utility itemset mining using length upper-bound reduction," in *Proc. Int. Conf. Ind., Eng. Other Appl. Appl. Intell. Syst.*, in Lecture Notes in Computer Science, vol. 9799. Springer, 2016, pp. 115–127.
[14] P. Fournier-Viger, J. C. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam, "The SPMF open-source data mining library version 2," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*, in Lecture Notes in Computer Science, vol. 9853. Springer, 2016, pp. 36–40.
[15] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas, "A survey of sequential pattern mining," *Data Sci. Pattern Recognit.*, vol. 1, no. 1, pp. 54–77, 2017.
[16] P. Fournier-Viger, C. Wu, S. Zida, and V. S. Tseng, "FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning," in *Proc. Int. Symp. Methodologies Intell. Syst.*, in Lecture Notes in Computer Science, vol. 8502. Springer, 2014, pp. 83–92.
[17] T. Gauthier and C. Kaliszyk, "Premise selection and external provers for HOL4," in *Proc. Conf. Certified Programs Proofs (CPP)*, 2015, pp. 48–57.
[18] T. Gauthier, C. Kaliszyk, and J. Urban, "TacticToe: Learning to reason with HOL4 tactics," in *Proc. LPAR*, in EPiC Series in Computing, vol. 46. 2017, pp. 125–143.
[19] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry, "A machine-checked proof of the odd order theorem," in *Proc. Int. Conf. Interact. Theorem Proving*, in Lecture Notes in Computer Science, vol. 7998. Springer, 2013, pp. 163–179.
[20] T. C. Hales *et al.*, "A formal proof of the Kepler conjecture," *CoRR*, vol. abs/1608.02644, 2015.
[21] O. Hasan and S. Tahar, "Formal verification methods," in *Encyclopedia of Information Science and Technology*, 3rd ed. Hershey, PA, USA: IGI Global, 2015.
[22] G. Irving, C. Szegedy, A. A. Alemi, N. Eén, F. Chollet, and J. Urban, "Deepmath-deep sequence models for premise selection," in *Proc. Neural Inf. Process. Syst. (NIPS)*, pp. 2243–2251, 2016.
[23] C. Kaliszyk, F. Chollet, and C. Szegedy, "HolStep: A machine learning dataset for higher-order logic theorem proving," *CoRR*, vol. abs/1703.00426, 2017.
[24] C. Kaliszyk, L. Mamane, and J. Urban, "Machine learning of Coq proof guidance: First experiments," in *Proc. SCSS*, in EPiC Series in Computing, vol. 30. 2014, pp. 27–34.
[25] C. Kaliszyk and J. Urban, "HOL(y)hammer: Online ATP service for HOL light," *Math. Comput. Sci.*, vol. 9, no. 1, pp. 5–22, Mar. 2015.
[26] E. Komendantskaya and J. Heras, "Proof mining with dependent types," in *Proc. Int. Conf. Intell. Comput. Math.*, in Lecture Notes in Computer Science vol. 10383. Springer, 2017, pp. 303–318.
[27] J. C.-W. Lin, J. Zhang, P. Fournier-Viger, T.-P. Hong, and J. Zhang, "A two-phase approach to mine short-period high-utility itemsets in transactional databases," *Adv. Eng. Informat.*, vol. 33, pp. 29–43, Aug. 2017.
[28] J. Liu, K. Wang, and B. C. M. Fung, "Direct discovery of high utility itemsets without candidate generation," in *Proc. IEEE 12th Int. Conf. Data Mining*, Dec. 2012, pp. 984–989.
[29] M. Liu and J. Qu, "Mining high utility itemsets without candidate generation," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2012, pp. 55–64.
[30] Y. Liu, W. Liao, and A. N. Choudhary, "A two-phase algorithm for fast discovery of high utility itemsets," in *Proc. Pacific-Asia Conf. Knowl. Discovery Data Mining*, in Lecture Notes in Computer Science, vol. 3518. Springer, 2005, pp. 689–695.
[31] M. S. Nawaz, M. Malik, Y. Li, M. Sun, and M. I. U. Lali, "A survey on theorem provers in formal methods," *CoRR*, vol. abs/1912.03028, 2019.
[32] M. S. Nawaz and M. Sun, "Reo2PVS: Formal specification and verification of component connectors," in *Proc. SEKE*, 2018, pp. 391–396.
[33] M. S. Nawaz and M. Sun, "Using PVS for modeling and verification of probabilistic connectors," in *Proc. Int. Conf. Fundam. Softw. Eng.*, in Lecture Notes in Computer Science, vol. 11761. Springer, 2019, pp. 61–76.
[34] M. S. Nawaz, M. Sun, and P. Fournier-Viger, "Proof guidance in PVS with sequential pattern mining," in *Proc. Int. Conf. Fundamentals Softw. Eng.*, in Lecture Notes in Computer Science, vol. 11761. Springer, 2019, pp. 45–60.
[35] M. Z. Nawaz, O. Hasan, M. S. Nawaz, P. Fournier-Viger, and M. Sun, "Proof searching in HOL4 with genetic algorithm," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, Mar. 2020, pp. 513–520.
[36] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert, "PVS system Guide, PVS prover Guide, PVS language reference," SRI Int., Menlo Park, CA, USA, Tech. Rep., Nov. 2001.
[37] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-mine: Hyper-structure mining of frequent patterns in large databases," in *Proc. IEEE Int. Conf. Data Mining*, Nov./Dec. 2001, pp. 441–448.
[38] *PVS and HUIM Data*. [Online]. Available: https://github.com/saqibdola/HUIM-PVS
[39] K. Slind and M. Norrish, "A brief overview of HOL_4," in *Proc. Int. Conf. Theorem Proving Higher Order Logics*, in Lecture Notes in Computer Science, vol. 5170. Springer, 2008, pp. 28–32.
[40] W. Song and C. Huang, "Mining high utility itemsets using bio-inspired algorithms: A diverse optimal value framework," *IEEE Access*, vol. 6, pp. 19568–19582, 2018.
[41] V. S. Tseng, C.-W. Wu, P. Fournier-Viger, and P. S. Yu, "Efficient algorithms for mining Top-K high utility itemsets," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 1, pp. 54–67, Jan. 2016.
[42] V. S. Tseng, C. Wu, B. Shie, and P. S. Yu, "UP-growth: An efficient algorithm for high utility itemset mining," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2010, pp. 253–262.
[43] T. Uno, M. Kiyomi, and H. Arimura, "LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets," in *Proc. FIMI Workshop*, 2004, pp. 1–11.
[44] F. Wiedijk. *Formalizing 100 Theorems*. Accessed: Apr. 3, 2020. [Online]. Available: https://cs.ru.nl/~freek/100
[45] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 3, pp. 372–390, May/Jun. 2000.
[46] S. Zida, P. Fournier-Viger, J. C. Lin, C. Wu, and V. S. Tseng, "EFIM: A highly efficient algorithm for high-utility itemset mining," in *Proc. Mexican Int. Conf. Artif. Intell.*, in Lecture Notes in Computer Science, vol. 9413. Springer, 2015, pp. 530–546.
[47] S. Zida, P. Fournier-Viger, C. Wu, J. C. Lin, and V. S. Tseng, "Efficient mining of high-utility sequential rules," in *Proc. Int. Workshop Mach. Learn. Data Mining Pattern Recognit.*, in Lecture Notes in Computer Science, vol. 9166. Springer, 2015, pp. 157–171.

M. SAQIB NAWAZ received the B.S. degree in computer systems engineering from the University of Engineering and Technology, Peshawar, Pakistan, in 2011, the M.S. degree in computer science from the University of Sargodha, Pakistan, in 2014, and the Ph.D. degree from Peking University, Beijing, China, in 2019. He is currently a Postdoctoral Fellow with the Center of Innovative Industrial Design (CIID), Harbin Institute of Technology, Shenzhen, China. His research interests include formal methods (theorem provers and model checkers), evolutionary computation, and the use of machine learning and data mining in software engineering.

PHILIPPE FOURNIER-VIGER is a Full Professor with the Harbin Institute of Technology (Shenzhen), China. Five years after completing his Ph.D. degree, he came to China and became a Full Professor with the Harbin Institute of Technology (Shenzhen), after obtaining a title of national talent from the National Science Foundation of China. He has published more than 250 research papers in refereed international conferences and journals, which have received more than 5200 citations. He is also the Founder of the popular SPMF open-source data mining library, which provides more than 170 algorithms for identifying various types of patterns in data. The SPMF software has been used in more than 800 articles, since 2010, for many applications from chemistry, smartphone usage analysis restaurant recommendation, to malware detection. He is the Editor of the book *High Utility Pattern Mining: Theory, Algorithms and Applications* (Springer, 2019), and a co-organizer of the Utility Driven Mining and Learning Workshop at KDD 2018 and ICDM 2019 and 2020. His research interests include data mining, frequent pattern mining, sequence analysis and prediction, big data, and applications.



JI ZHANG (Senior Member, IEEE) received the B.E. degree from the Department of Information Management and Information Systems, Southeast University, China, in 2000, the M.Sc. degree from the Department of Computer Science, National University of Singapore, in 2002, and the Ph.D. degree from the Faculty of Computer Science, Dalhousie University, Canada, in 2008. From 2008 to 2009, he was a Postdoctoral Research Fellow with the CSIRO ICT Center, Hobart, Australia. He is currently an Associate Professor in computing with the University of Southern Queensland (USQ), Australia. He has published over 140 articles in major peer-reviewed international journals and conferences. His research interests include big data analytics, knowledge discovery and data mining (KDD), and information privacy and security. He is a member of ACM, a Fellow of the Australian Endeavour and Queensland, Australia, and a Scholar of Izaak Walton Killam, Canada.