Discovering Implicit Redundancies in Network Communications for Detecting Inconsistent Values

Bogdan Tomoyuki Nassu, Takashi Nanya and Hiroshi Nakamura
Research Center for Advanced Science and Technology (RCAST)
The University of Tokyo, Japan
{bogdan,nanya,nakamura}@hal.rcast.u-tokyo.ac.jp

Abstract

Detecting inconsistent values received in a communication is a challenging problem faced in networked systems. Inconsistent values occur when a message contains incorrect data, even though the syntax is correct and there is no corruption due to transmission errors. In many cases, traditional schemes based on voting protocols or error detection codes cannot be used. An alternative is discovering implicit redundancies, or patterns that model a correct communication, and using these patterns to detect inconsistent values. However, existing techniques do not cover the inputs and sequential patterns needed by this problem. In this paper, we propose a novel technique that considers messages with multiple types and attributes, events involving variables, and a heuristic for reducing redundant information. Experiments show that the discovered redundancies can achieve reasonable error detection coverage in fields where sequential relations exist, without implying in a large number of false alarms or a high latency.

1. Introduction

In a network composed by multiple computer systems, a system communicates with other systems designed by different parties, in unknown conditions, and which possibly have design flaws that may cause interaction faults [3]. As a result, the system may end up receiving an inconsistent value — a type of error characterized by a message that contains incorrect data, even though the syntax is correct and there are no transmission errors. The data is syntactically correct, but inconsistent with what it should represent — for instance, a field called “User Name” containing a valid name that does not correspond to the actual user name.

Detecting inconsistent values is a challenging problem. Error detection codes and syntax checks cannot be used, as the message is generated with the inconsistent value, and is syntactically correct. Moreover, it is often not possible to modify a flawed system designed by a third-party, nor use unit duplication [11] or implement new voting protocols. An alternative is exploring implicit redundancies — patterns of behavior associated with seemingly unrelated events or data. It is common practice to use those redundancies informally, with the relations being identified in an ad hoc manner by a human expert who writes error detection functions based on his own knowledge about the problem domain. However, humans may commit mistakes and overlook relations, and automated techniques are needed.

The problem addressed in this paper is: based on examples of correct communications, discover rules that describe implicit redundancies that can be used to detect inconsistent values. There are no general methods for detecting inconsistent values; and though there are problems with similar characteristics, existing data mining techniques do not cover the inputs and sequential patterns needed for this problem. In this paper, we propose a new technique that explores characteristics of the problem domain in order to produce rules suitable for detecting inconsistent values. That includes considering messages with multiple types and attributes, sequences of events involving variables, and a heuristic for reducing redundant information. Experiments show that the discovered rules can achieve reasonable error detection coverage in fields where sequential relations exist, without implying in a large number of false alarms or a high latency.

This paper is organized as follows. In section 2, we discuss related work. In section 3, the problem is formalized. Section 4 describes our approach for discovering implicit redundancies and detecting inconsistent values. Section 5 describes the experiments performed to evaluate our approach. Section 6 concludes the paper.

2. Related Work

This general idea behind the proposed approach is discovering patterns that characterize a normal behavior and
using them to detect abnormal situations. Similar ideas were used previously in domains such as error detection in databases [12] and intrusion detection [6]. However, existing approaches are not suited to the addressed problem, as they do not consider sequential relations between discrete data with multiple types and attributes, working better for single-attribute time series, numeric data, and sequence-independent relations. To the authors’ knowledge, no other general methods for detecting inconsistent values were described in literature — inconsistent values are detected in an ad hoc manner, as traditional schemes involving temporal/spacial redundancies [11] are not able to detect them.

The sub-problem of discovering implicit redundancies can be seen as part of the broad temporal data mining field. The problem shares several characteristics with other work in the area, specially those related with generative models. A survey on temporal data mining can be found in [2].

The problem most closely related with our work is the discovery of frequent episodes in sequences of events, introduced in [8], and expanded in [9]. There are several similar approaches (e.g. [14, 4, 7]), which address different kinds of episodes and events. Our approach is based on the same general idea, a variant of the Apriori algorithm [1], where candidate episodes are generated from smaller frequent episodes. However, existing approaches do not consider the peculiarities of the problem of discovering implicit redundancies. For that reason, we propose new algorithms and adapt existing concepts. Major differences include:

- In many approaches, the input is a single sequence of events. Though some approaches do address the case of multiple-attribute inputs, they consider only events described as \((\text{attribute, value})\) pairs, while we use more complex types of events, such as storing values in variables and performing comparisons over them.
- We explore the fact that many protocols divide the messages in different types, while existing approaches make no distinction, for example, between an action response and an event notification.
- We do not specify time limits between events, considering unbounded episodes in a more general way than seen in previous work such as [4].
- While existing work discover rules based only on their frequency, we consider a stronger requirement that can be used to determine the expected values for a field.
- We include a heuristic for reducing the number of redundant rules based on their ability to detect errors.

### 3. Basic Definitions

In this section, the problem of locating implicit redundancies is formalized. We define the inputs (section 3.1) for this problem, and the concepts of events (section 3.2), episodes (section 3.3) and rules (section 3.4). We also define the episode prefix tree data structure (section 3.5). Note that, though many concepts can be found in previous work, we propose new definitions, as explained in section 2.

#### 3.1. Input

The input is a set \(\epsilon\), where each \(E \in \epsilon\) is a correct, noise-free communication example. A communication example \(E = \{M_1, ..., M_n\}\) is an ordered sequence of messages so that, for all \(i, j = 1, ..., n; M_i \in E\) appears before \(M_j \in E\) iff \(i < j\). The exact time messages were sent or received is not taken into account, only the sequence. A message \(M_i = \text{type}(\text{dir})\{(A_1, V_1), ..., (A_n, V_n)\}\) is defined by:

- A position \(i\) in the sequence.
- A type, specified by the protocol: e.g. “action response”, “query request”, “event notification”, etc.
- A direction: incoming or outgoing, relative to the system that will detect inconsistent values.
- A set of \((\text{attribute, value})\) pairs, where each attribute is a data field (e.g. “sender”, “seq”, “flag”, etc.). This set is denoted by \(AV = \{(A_1, V_1), ..., (A_n, V_n)\}\), and may be different for each message, or empty.

#### 3.2. Events

One or more events can occur in each message from the input, when a message contains attributes with specific values, or when the value in one of the message’s attributes is assigned or compared to a variable. An event \(e\) is represented as: \(e = \text{type}(\text{dir})\{T_e\}\), where \(\text{type}(\text{dir})\) is a message type, \(\text{dir}\) is in or out, and \(T_e = \{T_{e1}, ..., T_{en}\}\) is a set of tuples. Each tuple \(T_{ex} \in T_e\) is of one of three kinds:

- An \((\text{attribute, value})\) pair \((A_t, V_t)\).
- A “variable assignment”, represented by a pair \((A_t, X)\), where \(X\) is a variable.
- A “comparison with variable”, represented by a triple \((A_t, \circ, X)\), where \(\circ \in \{=, \sim, \geq, \leq, \subseteq, \supseteq\}\), and \(X\) is a variable with an assigned value.

We consider only events with at most one variable assignment or comparison with variable. Though the number of interesting events may be reduced, that restricts the search space, making the discovery of rules feasible.

Let \(S\) be the set of all strings, \(\mathbb{R}\) the set of real numbers, and \(\mathbb{B}\) the boolean set \(\{true, false\}\). Assuming a variable \(X\) contains values from any of the above sets, given a message \(M = \text{type}(\text{dir})\{AV_m\}\) so that \(AV_m = \{(A_u, V_a), ..., (A_b, V_b)\}\), event \(e\) occurs in \(M\) if:
• type\(_m\) = type\(_e\) and dir\(_m\) = dir\(_e\)
• If T\(_e\) = \(\emptyset\), AV\(_{m} = \emptyset\).
• If T\(_e\) \(\neq \emptyset\), for all tuples T\(_{et}\) \(\in T_e\):
  
  - If T\(_{et}\) = (A\(_1\), V\(_1\)), T\(_{et}\) \(\in AV_m\), i.e. M has an attribute A\(_1\) with value V\(_1\).
  
  - If T\(_{et}\) = (A\(_1\), X), A\(_1\) \(\in \{A_a, ..., A_b\}\), i.e. M has attribute A\(_1\). After the event occurs, any comparisons over variable X are actually performed over value V\(_1\).
  
  - If T\(_{et}\) = (A\(_t\), ∅, X), A\(_t\) \(\in \{A_a, ..., A_b\}\) and V\(_t\) \(\cap X\) is true. V\(_t\) \(\cap X\) is true if:
    * If \(\circ \in \{\{\}\}\), V\(_t\) = X.
    * If \(\circ \in \{\sim\}\), V\(_t\) \(\in \mathbb{R}\), and V\(_t\) is true and X is false, or vice-versa.
    * If \(\circ \in \{\ge\}\), V\(_t\) \(\in \mathbb{R}\), and V\(_t\) \(\ge\) X.
    * If \(\circ \in \{\le\}\), V\(_t\) \(\in \mathbb{R}\), and V\(_t\) \(\le\) X.
    * If \(\circ \in \{\sim\}\), V\(_t\) \(\in \mathbb{R}\), and V\(_t\) is a substring of X.
    * If \(\circ \in \{\le\}\), V\(_t\) \(\in \mathbb{R}\), and X is a substring of V\(_t\).

For instance, given message M = t(in)\(\{(A, true)\}\), and variable X = false, events t(in)\(\{(A, true)\}\), t(in)\(\{(A, X)\}\) and t(in)\(\{(A, \sim, X)\}\) occur in M.

3.3. Episodes

An episode \(\alpha = \{e_1, ..., e_n\}\) is an ordered sequence of events so that for all i, j \(= 1, ..., n\); event e\(_i\) \(\in \alpha\) occurs before event e\(_j\) \(\in \alpha\) if i < j. We call |\(\alpha\)| the length of \(\alpha\).

An episode can be represented as a sequence of events with the \(\rightarrow\) symbol indicating the order: \(\alpha = e_1 \rightarrow \ldots \rightarrow e_n\).

An episode must have at most one event with a variable assignment (A\(_{et}\), X) and one event with a comparison (A\(_{jt}\), ∅, X). If both are present, the assignment must come before the comparison, and a value is assigned to variable X during the former (i.e. the comparison is V\(_{t}\) \(\cap V\(_{e}\)\)). Though some interesting episodes may not be discovered, the search space is reduced, making the discovery of rules feasible.

An episode \(\alpha\) occurs in a sequence of messages E if:

1. There is an injective mapping \(o : \alpha \rightarrow E\) so that every e\(_i\) \(\in \alpha\) occurs in a M\(_k\) \(\in E\) (i.e. each event in an episode occurs in a message from the sequence).

2. For all i, j \(= 1, ..., |\alpha|\), given o(e\(_i\)) = M\(_k\) and o(e\(_j\)) = M\(_l\), if i < j then k < l (i.e. events occur in the sequence in the same order as they appear in the episode).

3. Given e\(_i\), e\(_j\) \(\in \alpha\) so that o(e\(_i\)) = M\(_k\), o(e\(_j\)) = M\(_l\), and 1 \(\leq i, j \leq |\alpha|\); if e\(_j\) is associated with message type t and direction d, there is no M\(_m\) associated with t(d) so that k < m < l, unless there exists e\(_n\) \(\in \alpha\) so that i < n < j and o(e\(_n\)) = M\(_m\) (i.e. if an event associated with t(d) occurs, no other message of type t and direction d may appear before the last event in the episode occurs, unless there is another event in the episode also associated with t(d)).

Condition 3 restricts the search space by exploring the fact that messages from the same type are related. This provides a heuristic bias to our approach, and though interesting episodes may be ignored, rule discovery becomes more efficient. For example, if a request is followed by a response, responses of subsequent requests should be ignored, unless the second request is also in the episode. Hence, episode REQ(out)\(\{(A_1, V_1)\}\) \(\rightarrow\) RES(in)\(\{(A_2, V_2)\}\) does not occur in the sequence below:

\[
\text{REQ(out)}\{\{(A_1, V_1)\}\}; \text{RES(in)}\{\{(A_2, V_3)\}\}; \\
\text{REQ(out)}\{\{(A_1, V_3)\}\}; \text{RES(in)}\{\{(A_2, V_2)\}\}
\]

The first and second events occur, respectively, in the first and fourth messages, satisfying conditions 1 and 2. However, the third message has the same type and direction as the first event, so condition 3 is not satisfied.

An episode can occur several times in a sequence, with different mappings that satisfy the restrictions above. An occurrence of \(\alpha = e_1 \rightarrow \ldots \rightarrow e_n\) is minimal if \(\alpha\) occurs in sequence E with o\(_1\)\(\{e_1\}\) = M\(_k\) and o\(_2\)\(\{e_n\}\) = M\(_l\), and there is no other mapping for \(\alpha\) in E so that o\(_2\)\(\{e_1\}\) = M\(_k\), o\(_2\)\(\{e_n\}\) = M\(_l\) and i < k, l < j.

3.3.1 Sub-Episodes.

An episode \(\beta = e_{b_1} \rightarrow \ldots \rightarrow e_{b_n}\) is a sub-episode of \(\alpha = e_{a_1} \rightarrow \ldots \rightarrow e_{a_n}\) if |\(\beta\)| < |\(\alpha\)| and there is an injective mapping \(f : \beta \rightarrow \alpha\) such that for all e\(_b\) \(\in \beta\), f(e\(_b\)) \(\in \alpha\) and if f(e\(_{b_i}\)) = e\(_{a_j}\), then f(e\(_{b_{i+1}}\)) = e\(_{a_{j+1}}\), for all e\(_b\) \(\neq e_{b_n}\). For example, \(\alpha = e_1 \rightarrow e_2 \rightarrow e_3\) has the following sub-episodes: e\(_1\), e\(_2\), e\(_3\), e\(_1\) \(\rightarrow\) e\(_2\) and e\(_2\) \(\rightarrow\) e\(_3\). Note that by the above definition, e\(_1\) \(\rightarrow\) e\(_3\) is not a sub-episode of \(\alpha\).

3.4. Rules

An episode \(\alpha = e_1 \rightarrow \ldots \rightarrow e_n\) is a rule if it satisfies the following conditions:

1. \(\alpha\) is frequent: the number of minimal occurrences of \(\alpha\) in the input \(\epsilon\) is greater than or equal to a given threshold, or minimum support.

2. Given i < k < j, and event e\(_n\) associated with message type t and direction d, if e\(_1\) \(\rightarrow \ldots \rightarrow e_{n-1}\) occurs with o(e\(_n\)) = M\(_i\), and e\(_n\) occurs in a message M\(_j\) associated with t(d), there is no M\(_k\) associated with t(d).

3. If \(\alpha\) has events involving variables, there is one e\(_i\) with a variable assignment (A\(_{it}\), X) and one e\(_j\) with a comparison (A\(_{jt}\), ∅, X), with i < j: no rule can have a value assigned to a variable with no comparisons over it.
Informally, a rule is a pattern that arises frequently, and that is not contradicted by the input: every time events $e_1$ to $e_n$ occur, $e_n$ occurs in the first message of type $t$ and direction $d$ after $o(e_{n-1})$. $e_n$ is the consequent of the rule. As errors are detected in received messages, we consider only rules with the consequent referring to incoming messages.

### 3.5. Episode Prefix Trees

Prefix trees [5], are a classic data structure used for pattern matching. Looking for the occurrence of an episode in a sequence of messages is essentially a pattern matching problem with additional constraints. Thus, the verification can be accelerated by encoding a set of episodes as an episode prefix tree (EPT).

Each path from the root to a node in an EPT defines an episode, and each branch an event. All the descendants of a node in an EPT share a common prefix (i.e. start with the same events), defined by the sub-episode from the root to $N$. An EPT can be a compact representation for a set of episodes consisting of a single event. Then, during each iteration, larger candidates are created based on smaller frequent episodes identified previously, and frequent episodes and rules are located among the candidates. An episode prefix tree is used to represent the candidate episodes.

![Figure 1. An episode prefix tree.](image)

### 4. Discovering Implicit Redundancies

This section describes our approach for discovering rules. The discovery algorithm is described in section 4.1. A cross-checking process (section 4.2) is used to avoid false discoveries and overfitting. Section 4.3 shows how the rules can be used for detecting errors. Section 4.4 describes an heuristic for reducing the number of redundant rules.

#### 4.1. Rule Discovery Algorithm

A brute-force approach for discovering implicit redundancies would consist of generating all possible sequences of events and checking if each “candidate” sequence satisfies the conditions for being a rule. Such an approach can result in a combinatorial explosion of candidates, given the number of possible combinations of events. For that reason, we employ a series of techniques, in order to minimize the number of considered events and candidates, reduce the set of rules, and accelerate its verification.

The algorithm is built around the well known premise, used by previous work (e.g. [1, 9, 14]), that an episode can only be frequent if all of its sub-episodes are also frequent. Thus, instead of testing the entire search space, we may avoid generating candidate episodes with non-frequent sub-episodes. The process is iterative, and begins with a set of episodes consisting of a single event. Then, during each iteration, larger candidates are created based on smaller frequent episodes identified previously, and frequent episodes and rules are located among the candidates. An episode prefix tree is used to represent the candidate episodes.

Algorithm 1 receives three parameters: $\epsilon$ is the set of input examples, $\text{minsupport}$ is the minimum support, and $\text{maxlen}$ is the maximum number of events in an episode. The $\text{maxlen}$ parameter is provided solely for performance reasons: we may wish to stop rule discovery before episodes become too long or too numerous. Below, we discuss in more detail several aspects of this algorithm.

**Algorithm 1 Rule discovery algorithm.**

```plaintext
learnRules ($\epsilon$, $\text{minsupport}$, $\text{maxlen}$)
    $EV \leftarrow$ events that occur in the messages in $\epsilon$
    $EPT \leftarrow$ EPT with one child node for each $e \in EV$
    $ncandidates \leftarrow |EV|$
    $R \leftarrow \emptyset$
    for each invariant and incoming $e \in EV$ do
        $R \leftarrow R \cup \{e\}$
    end for
    for $L = 1$; $L < \text{maxlen}$ AND $ncandidates > 0$; $L += 1$ do
        $ncandidates \leftarrow \text{expandEpisodePrefixTree (EPT, L)}$
        if $ncandidates > 0$ then
            $\text{checkOCCurrences (e, EPT, L + 1, \text{minsupport})}$
            $\text{removeNotFrequent (EPT’s root node, \text{minsupport})}$
            $\text{remove from EPT subtrees that don’t go down to level L}$
            $\text{checkRules (EPT’s root node, R, L + 1)}$
        end if
    end for
    return $R$
```

#### 4.1.1 Choosing Events

Candidate episodes are built by combining events that occur in the input messages. Considering all possible combinati-
ons of attributes, values, variables and comparisons can lead to an explosion in the search space: in a message with \( a \) attributes, \( \sum_{p=1}^{a} C_p + \sum_{q=1}^{n!} \frac{(q-1)(n-q)!}{q!} \) events can occur. For example, 3263 events can occur in a message with 7 attributes. Therefore, the number of considered events must be reduced, as long as no relevant information is lost.

Events involving variables are more numerous than those containing only \((\text{attribute, value})\) pairs: for each attribute, there are at least 7 events involving variables, plus combinations of these events with other \((\text{attribute, value})\) tuples. To reduce this number, if an event has a variable associated with an attribute \( A \) that has always the same value, the event is discarded. Otherwise, we represent all the tuples containing variables associated with \( A \) as a a singleton \([A]\). For instance, events \( t(d)\setminus\{(A_1, X), (A_2, V_2)\} \) and \( t(d)\setminus\{(A_1, \sim X), (A_2, V_2)\} \) are both represented as \( t(d)\setminus\{(A_1), (A_2, V_2)\} \). This representation is possible because all the events in an episode refer to a single variable, and the type of event (assignment or comparison) can be decided based on its position in the episode and on the successful comparisons observed previously. Thus, for a message with \( a \) attributes, the number of events can be reduced to \( \sum_{p=1}^{a} C_p + \sum_{q=1}^{n!} \frac{(q-1)(n-q)!}{q!} \). That represents, for example, a reduction from 3263 to 575 events in a message with 7 attributes. Note this is just a result of using a compact representation for the same tuples, and no information is lost.

We also discard redundant events: if \( e_1 = t(d)\setminus T_1 \) occurs in the same messages as \( e_2 = t(d)\setminus T_2 \) and \( T_1 \supset T_2 \), \( e_2 \) can be discarded. This results in less candidates being generated, but no information is lost: for every non-discovered episode with \( e_2 \) as its \( i \)-th event, there will be a corresponding discovered episode with \( e_1 \) as the \( i \)-th event.

Finally, we use the algorithm similar to Apriori [1] presented in [14] to locate frequent subsets of attributes and values. That algorithm is based on the observation that no frequent episode can contain events which are not frequent. This results in a smaller set of events, but the discovered rules remain the same: the episodes containing non-frequent events are removed in the first iteration of algorithm 1.

### 4.1.2 Candidate Generation

During each iteration, to expand the EPT and generate new candidates, for each leaf node \( N \) with a path defining a frequent episode \( e_{L-1}\rightarrow \ldots \rightarrow e_1 \), we take all paths defining frequent episodes \( f_{L-1}\rightarrow \ldots \rightarrow f_i \) so that \( f_i = e_{i+1} \) for all \( 1 \leq i \leq L - 2 \). Then, if that does not result in an episode with more than two events involving variables, one child node is added after \( N \) for each \( f_{L-1} \) found. The new candidate episodes will have the leaves at level \( L \), and all of their sub-episodes will be frequent.

### 4.1.3 Finding Frequent Episodes

After candidate episodes are generated in the EPT, their occurrences are counted. Checking if an event occurs in a message is an expensive operation: all the event’s attributes must be located in the message, and the (string) values must be compared. However, once the events that occur in a message are known, those comparisons can be avoided, for events not involving variables, which are equivalent to comparisons with constants. Each message is then associated with a set of references to the events not involving variables that occur in it. These references require memory space, but the occurrence of events can be checked faster.

To quickly locate frequent episodes, we also explore the EPT data structure. Algorithm 2 finds frequent candidate episodes and rules in an EPT with \( L \) levels. With a single pass through the input messages, all the candidates are tested, based on pointers to nodes in the tree. Those pointers \((N,V)\) represent the sub-episodes observed in the input, possibly including a variable with an assigned value. For each message in the input, the list of pointers is updated. There is also a list of pointers to the nodes in the first level of the EPT \((ROOT, NV)\), which are checked for all the messages.

**Algorithm 2 Finding frequent episodes using an EPT.**

```plaintext
checkOccurrences (\( \epsilon, ept, L, \text{minsup} \))
ROOT_NV \leftarrow \emptyset
for each child node \( N \) of ept’s root node do
    ROOT_NV \leftarrow ROOT_NV \cup \{(N,[\text{newvariable}])\}
end for
for each example \( E \in \epsilon \) do
    NV \leftarrow \emptyset
    for each message \( M \in E \) do
        NEXT \leftarrow \emptyset
        for each pair \((N,V)\) \in ROOT_NV do
            checkNode ((N,V), M, L, \text{minsup}, NEXT)
        end for
        for each pair \((N,V)\) \in NV do
            if checkNode ((N,V), M, L, \text{minsup}, NEXT) then
                NV \leftarrow NV \cup \{(N,V)\}
            end if
        end for
        NV \leftarrow NV \cup NEXT
    end for
end for
```

Algorithm 3 is used to check each node. It receives as parameters a pair \((\text{node, variable})\), a message \( M \), and a set \( \text{NEXT} \) of nodes that will be checked against the next message. Besides checking if the episode ending in \( N \) occurs in the sequence, this algorithm also checks which leaf nodes can be rule consequents. If the event associated with \( N \) is a variable assignment, the value of the corresponding variable is stored. If the event associated with \( N \) is a variable comparison, each successful comparison indicates one
event, and if different comparisons are successful, \( N \) cannot be the consequent of a rule. It returns \( \text{false} \) if \( N \) must be checked in the next iteration of algorithm 2, \( \text{true} \) otherwise.

**Algorithm 3 Checking a node.**

```plaintext
checkNode \(((N, V), M, L, \text{minsup}, \text{NEXT})\)
if there is 1 event involving variable from the root to \( N \) then
  \( N \) cannot be the consequent of a rule
end if
if the event associated with \( N \) has a different type than \( M \) then
  if \( N \) has an ancestor with the same type as \( M \) then
    return \text{true}
  else
    return \text{false}
  end if
end if
if the event \( e \) associated with \( N \) occurs, considering \( V \) then
  if the comparisons were different than previous successful comparisons made in \( N \) then
    \( N \) cannot be the consequent of a rule
  else if \( e \) is a variable comparison event then
    \( V \leftarrow \) the value of the attribute involving variables in \( e \)
  end if
if \( N \) is a leaf node then
  increase number of occurrences of \( N \) by 1
else
  for each child node from \( N \) in a subtree with \( L \) levels do
    \( \text{NEXT} \leftarrow \text{NEXT} \cup \{(\text{child}, V)\} \)
  end for
end if
else
  \( N \) cannot be the consequent of a rule
end if
return \text{true}
```

### 4.1.4 Checking Rules

After isolating the frequent candidates, algorithm 1 finally checks for rules with length \( L + 1 \) in the EPT. The rules are chosen based on the information collected by algorithm 3. Not all rules have to be considered: in several cases, different rules containing the same information are discovered. We use the following criteria to discard redundant rules:

- Given event \( t(d)\backslash \{(A_1, V_1), ..., (A_n, V_n)\} \in \text{EV} \) containing only \((\text{attribute}, \text{value})\) pairs, we look for event \( t(d)\backslash \{(A_1, V_2), ..., (A_n, V_2)\} \in \text{EV} \) so that for at least one \( i \), \( V_1 \neq V_2 \). If there is no such event, attributes \( A_1, ..., A_n \) are invariant, and always have the same value for events of type \( t \) and direction \( d \). We may discard all rules with length greater than 1 which have an invariant event as their consequent, as it is always true, independent of the events which occur before it.

- If there are at least two rules \( \alpha = e_1 \rightarrow ... \rightarrow e_n \) and \( \beta = f_1 \rightarrow ... \rightarrow f_m \rightarrow e_1 \rightarrow ... \rightarrow e_n \) (i.e. \( \beta \) ends with the same events as \( \alpha \)), \( \beta \) is redundant, as its consequent is independent of events \( f_1, ..., f_m \).

- Given rules \( r_1 = e_1 \rightarrow ... \rightarrow e_n \) with no events involving variables, and \( r_2 = f_1 \rightarrow ... \rightarrow f_m \) with events involving variables, \( r_1 \) is more specific than \( r_2 \) if:
  - For all \( f_i \in r_2 \) not involving variables, \( e_i = f_i \).
  - Given the event \( f_j = t(d)\backslash T_{fj} \in r_2 \) so that there is a tuple \((A_a, X) \in T_{fj}, \) and \( e_j = t(d)\backslash T_{e_j} \in r_1 \), there is a tuple \((A_a, V_a) \in T_{e_j} \).
  - Given \( f_k = t(d)\backslash T_{fk} \in r_2 \) so that there is a tuple \((A_b, \odot, X) \in T_{fk}, \) and \( e_k = t(d)\backslash T_{ek} \in r_1 \), there is a tuple \((A_b, V_b) \in T_{ek} \) so that \( V_b \odot V_a \) is true.

All the sequences identified by the more specific rule are also identified by the more general rule. Thus, the more specific rule \( (r_1) \) can be discarded.

### 4.2. Cross-Checking

The minimum number of occurrences of each rule in the input is defined by the minimum support. Using a low minimum support can lead to false discoveries. Using a high minimum support can increase the confidence, but on the other hand, interesting rules can remain undiscovered. The over-specialization of the rules to the learning examples (overfitting) must also be taken into account.

To deal with the above problems, we employ a cross-checking process when learning rules. Implicit redundancies are located independently in \( S \) sets of learning examples. The rules located in each set are then compared to the other \( S - 1 \) sets, and if a rule is contradicted by any of the examples, it is discarded. The resulting \( S \) sets of rules are merged, and compared to a test set. If a rule is contradicted or does not occur in the test set, it is discarded.

### 4.3. Detecting Inconsistent Values

After implicit redundancies are discovered, they can be used to detect errors. The set of rules is represented by a single EPT that does not change while errors are being detected. The error detector receives a sequence of messages. We consider online error detection, meaning that errors in a message \( M_i \) are detected based on information present in a message \( M_j \) so that \( i \leq j \). This is done by a modified version of algorithm 2. In our implementation, the error detector produces error logs, used to evaluate the approach. Though it is possible to extend our approach to include online learning of rules, that raises other issues (e.g. faulty/malicious systems), and we do not address such situation in this paper.
4.4. Reducing Redundant Rules

In this section, we propose a technique for “simplifying” a rule set by reducing the number of redundant rules. This technique differs from the ones presented in section 4.1.4 because, instead of exploring general properties, it relies on a measure of the usefulness of the rules for detecting inconsistent values. The rule set is used to detect errors in a set of examples containing injected inconsistent values. Each rule is then associated with a list of errors it was able to detect, and by comparing those lists, it is possible to identify a subset of rules that can detect the same errors as the original rule set. The optimum subset of rules is the one that results in the smallest episode prefix tree, while keeping the same error detection capabilities from the original rule set.

Finding the optimum subset of rules is a challenging problem. If the injected faults are not representative of the actual faults, the simplified rule set may lose useful information. Besides, a set with $R$ rules has $\sum_{p=1}^{R} C_p^R$ possible subsets, and testing all the subsets can be unfeasible. For that reason, we adopt a heuristic method, which does not guarantee the optimum solution, but that can provide satisfactory results, as shown in section 5.

Algorithm 4 simplifies a set of rules. The errors detected by each rule in the faulty examples are identified by the $\text{errorsDetectedBy}$ function. Each rule is associated with a “cost per error”, given by the length of the rule (the number of events) divided by the number of errors detected by that rule. The cost per error is the bias that guides the heuristic, favoring short rules that detect many errors over longer rules that detect less errors.

**Algorithm 4 Removing redundant rules.**

```plaintext
simplify (rules)
simplified ← ∅
for each $r_1 \in$ rules do
    if errorsDetectedBy ($r_1$) ≠ ∅ then
        keep ← true
        for each $r_2 \in$ rules so that $r_1 \neq r_2$ do
            if costPerError ($r_1$) ≤ costPerError ($r_2$) AND errorsDetectedBy ($r_1$) ∪ errorsDetectedBy ($r_2$) then
                rules ← rules − $r_2$
            else if costPerError ($r_2$) < costPerError ($r_1$) AND errorsDetectedBy ($r_2$) ∪ errorsDetectedBy ($r_1$) then
                keep ← false
                get the next $r_1$
        end if
    end for
    if keep = true then
        simplified ← simplified ∪ $r_1$
    end if
end if
return simplified
```

5. Experimental Evaluation

In this section, we describe the experimental environment (section 5.1), and experiments that evaluate the proposed approach when used for detecting inconsistent values, in terms of error detection coverage, false alarm rate, and latency (respectively, sections 5.2, 5.3, 5.4 and 5.5).

5.1. Experimental Environment

The proposed approach was implemented in Java, with experiments being performed on a Windows-based PC, with 2GB memory and 1.6 GHz processor. We consider two devices running over the Universal Plug-and-Play (UPnP) [13] standard — an architecture for service exchange between electronic devices. The first is a simple Test device that controls a true/false flag, accepting control messages and requests to switch or set the flag, query for its value, or call for an action that has no effect at all. The second device is a Content Server with user access control — files can be stored in directories, copied, have their properties changed, etc. There are 87 fields where errors can occur in messages received from the Test device, 123 fields for the Content Server. Both devices run over a simulated environment that includes a real implementation of UPnP [10].

For each device, 5 sets of learning examples and one set of test examples were randomly generated. For the Test device, these sets had, respectively, a total of 360,404 and 191,856 messages. For the Content Server, the sets had 415,145 and 234,244 messages. These are the examples used to discover rules, as shown in section 5.2.

A number of faulty examples was also generated, with injected inconsistent values such as random values, values similar to the correct ones, values observed in the same field, etc. For the Test device, two sets of faulty examples were used: one containing errors in 12 fields, and another with errors in 73 fields (these numbers are explained in section 5.2). For the Content Server, the set of faulty examples contains errors in 79 fields. As the actual error distribution is unknown, the errors are randomly distributed, and the examples have different error rates (the number of injected errors divided by the number of messages). In all cases, we also consider a number of examples with no injected errors. Table 1 shows the number of examples and messages in each set of faulty examples, as well as its size.

<table>
<thead>
<tr>
<th>Device</th>
<th>Examples</th>
<th>Messages</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test (12-attribute)</td>
<td>8,820</td>
<td>6,423,900</td>
<td>3.01</td>
</tr>
<tr>
<td>Test (73-attribute)</td>
<td>35,280</td>
<td>25,354,663</td>
<td>11.91</td>
</tr>
<tr>
<td>Content Server</td>
<td>34,020</td>
<td>22,993,005</td>
<td>14.24</td>
</tr>
</tbody>
</table>

Table 1. Faulty Examples.
5.2. Rule Sets

Several sets of rules were discovered, using the cross-checking process and the learning and test examples. Two sets of rules were discovered for the Test device, using different values for the minimum support (described in section 3.4). For the first set of rules, the minimum support was chosen for each of the 5 sets of learning examples so that no rules are discarded by the cross-check, being between 1,380 and 5,000. For the second set, the minimum support was chosen so that up to 25% of the rules are discarded by the cross-check, being between 560 and 2,380. Those sets of rules, are called, respectively, the “non-contradicted” and the “cross-check” rule sets. For the Content Server, the minimum support is between 400 and 1,220, with only the “cross-check” rule set being discovered. In all cases, the maximum rule length was 10.

The heuristic from section 4.4 was used to simplify the cross-check rule sets, considering the errors detected in a subset of faulty examples containing only those with the highest error rates. For the Test device, that corresponds to 3.59% of the messages from the 73-attribute faulty examples, and for the Content Server, 4.53% of the messages.

Table 2 shows the number of rules in each set, the number of rules removed during cross-check (if applicable), and the number of covered fields, which is determined by the attributes present in rule consequents. In the faulty examples from section 5.1, errors are only injected in those fields.

<table>
<thead>
<tr>
<th></th>
<th>Rules</th>
<th>Covered Fields</th>
<th>Removed Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test (non-contradicted)</td>
<td>2,308</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Test (cross-check)</td>
<td>4,374</td>
<td>73</td>
<td>1,110</td>
</tr>
<tr>
<td>Test (simplified)</td>
<td>109</td>
<td>73</td>
<td>—</td>
</tr>
<tr>
<td>Content Server cross-check</td>
<td>27,978</td>
<td>79</td>
<td>8,990</td>
</tr>
<tr>
<td>Content Server simplified</td>
<td>127</td>
<td>79</td>
<td>—</td>
</tr>
</tbody>
</table>

The five obtained rule sets were used to detect errors in the faulty examples. If the cross-checking process works, the cross-check rule set for the Test device will have a better coverage than the non-contradicted rule set, without increasing the false alarm rate. If the heuristic for removing redundant rules works, the error detection latency will be reduced, without a heavy impact over the error detection coverage or the number of false alarms.

5.3. Error Detection Coverage

We consider the error detection coverage is the percentage of injected errors that are correctly detected. Different from traditional techniques based on explicit redundancies, our approach detects errors using only implicit relations that are already present in the communication. Moreover, inconsistent values can be hard to detect, appearing as syntactically correct values. For that reason, the coverage we expect to obtain is not as high as what can be obtained in other settings by schemes such as error detection codes or voting techniques. On the other hand, such techniques cannot be used to detect the errors addressed in this work at all. There is a limit to the coverage that can be achieved without adding redundant information to the communication, and judging whether the obtained coverage is high enough depends on specific application and dependability requirements.

The coverage that can be obtained depends on how the rules that describe implicit redundancies are defined. Though rules defined in a more general way could possibly detect more errors, that raises issues about how those rules could be learned (e.g. some definitions imply in an infinite search space). The coverage is also affected by the learning examples and the used minimum support — if a relation does not occur frequently in the learning examples, it will not be discovered. Finally, as our approach is based on sequential rules, an error may be detected or not, depending on whether the sequence of messages observed before the error led to a rule consequent or not.

The coverage is limited by the consequents of the rules in a set. Some attributes are not present in any rule consequent, either because the learning examples contained no frequent relations concerning that attribute, or because no such relations exist at all. No errors can be detected in those attributes. In the Test device, there are 87 attributes where errors can occur. The non-contradicted rule set has consequents associated with 12 of those attributes, meaning relations were found for 13.79% of the attributes. The cross-check rule set, as well as its simplified version, has consequents associated with 73 attributes, or 83.91% of them. The Content Server can have errors in up to 123 fields, of which 79 (64.23%) are covered by the obtained rules.

Table 3 shows the percentage of injected errors detected by each rule set in different sets of faulty examples. It can be seen that the cross-check rule set for the Test device has detected more errors than the non-contradicted rule set. That indicates the lower values used for the minimum support resulted in sets of rules with more useful information.

The simplified rule set for the Test device was able to detect almost all the errors detected by the original rule set — 99.996% of the errors in the 12-attribute faulty examples, and 99.999% for the 73-attribute examples. That indicates the simplified rule set contained only slightly less relevant information than the original set, even though it had only 2.49% of the original rules. For the Content Server, the simplified rule set had 0.45% of the original rules, but was able to detect 97.8% of the errors detected by the original.
rule set. The simplified rule sets have lower coverage, and that shows the heuristic did not remove only irrelevant rules. Even so, the loss in coverage is small compared to the size reduction obtained by the heuristic.

Table 3. Error detection coverage.

<table>
<thead>
<tr>
<th>Rule Set/Faulty Examples</th>
<th>Detected Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test (non-contradicted) 12-attribute faulty examples</td>
<td>79.91%</td>
</tr>
<tr>
<td>Test (cross-check) 12-attribute faulty examples</td>
<td>84.769%</td>
</tr>
<tr>
<td>Test (cross-check) 73-attribute faulty examples</td>
<td>83.624%</td>
</tr>
<tr>
<td>Test (simplified) 12-attribute faulty examples</td>
<td>84.766%</td>
</tr>
<tr>
<td>Test (simplified) 73-attribute faulty examples</td>
<td>83.622%</td>
</tr>
<tr>
<td>Content Server (cross-check) 73-attribute faulty examples</td>
<td>80.27%</td>
</tr>
<tr>
<td>Content Server (simplified)</td>
<td>78.51%</td>
</tr>
</tbody>
</table>

Table 4. False alarm rates.

<table>
<thead>
<tr>
<th>Rule Set/Faulty Examples</th>
<th>False Alarm Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test (non-contradicted) 12-attribute faulty examples</td>
<td>0.151%</td>
</tr>
<tr>
<td>Test (cross-check) 12-attribute faulty examples</td>
<td>0.151%</td>
</tr>
<tr>
<td>Test (cross-check) 73-attribute faulty examples</td>
<td>0.045%</td>
</tr>
<tr>
<td>Test (simplified) 12-attribute faulty examples</td>
<td>0.149%</td>
</tr>
<tr>
<td>Test (simplified) 73-attribute faulty examples</td>
<td>0.044%</td>
</tr>
<tr>
<td>Content Server (cross-check) 73-attribute faulty examples</td>
<td>0.11%</td>
</tr>
<tr>
<td>Content Server (simplified)</td>
<td>0.08%</td>
</tr>
</tbody>
</table>

5.4. False Alarms

A false alarm occurs if an error that did not occur is detected. There are two reasons why a false alarm may occur. One possible reason is that the used set of rules contains false discoveries or suffers from overfitting. The other possible cause for false alarms is an error that was not detected previously. As our approach is based on sequential rules, if an inconsistent value is not detected, some events that should be ignored will occur. Afterwards, when a correct value is received, it can be considered erroneous by some rule that was triggered by the undetected inconsistent value.

Table 4 shows the proportion of messages from the sets of faulty examples that have triggered false alarms, when using each rule set. In all cases, there were no false alarms when no errors were injected. The overfitting problem was not entirely avoided, though: in some cases, the injected errors made the protocol to generate rare exception messages, and those messages, while correct themselves (the protocol specifies them) were detected as being inconsistent. That happens because those messages are rare enough not to appear in the examples used for discovering rules. Note, however, that even though some false alarms did occur due to overfitting, they were all caused by a previous error that has actually occurred but was not detected. When grouping the faulty examples by their error rates, the correlation coefficient between the number of undetected errors and the number of false alarms was always higher than 0.9.

The cross-check rule set for the Test device has triggered almost the same number of false alarms as the non-contradicted rule set (the specific messages that have triggered them are different, though). That indicates the cross-checking process was able to compensate for the lower minimum support used when learning those rules. It can also be seen that the simplified rule sets have triggered less false alarms than the original sets.

5.5. Error Detection Latency

The latency, or the time needed for testing a message for the presence of errors, is crucial for online error detection. The exact latency depends on factors such as the protocol, platform, implementation, sequence of messages and the set of rules. One determinant factor is the number of nodes in the EPT containing the rules used for error detection, as errors are detected based on visits to these nodes.

Table 5 shows the number of nodes in the EPTs, the maximum and average number of node visits, and the average time for checking a message, in ms. The time can be compared in this case because we use the same implementations running on the same platform (but the results for the Test device cannot be compared with those from the Content Server). It can be seen that visiting more nodes in average leads to a higher error detection time. When grouping the faulty examples by their error rates, the correlation coefficient between the number of node visits and average error detection time was always higher than 0.9327. The maximum number of node visits for a message is limited by the number of nodes in the EPT. It is expected that, for a given protocol and sequence of messages, a smaller EPT will result in less node visits, and in a smaller latency. For the Test device, this is confirmed by the higher latency obtained by the cross-check rule set, which is larger than the non-contradicted rule set.

The number of EPT nodes and the error detection time is also considerably smaller for the simplified rule sets. The maximum rule length was reduced from 10 to 6 for the Test device, and from 10 to 7 in the Content Server; indicating
that long rules tend to be redundant and contain the same information as other, shorter rules. The tradeoff of using such an heuristic method is that the coverage is also reduced, but very slightly if compared to the gain in terms of latency.

<table>
<thead>
<tr>
<th>Rule Set/ Faulty Examples</th>
<th>EPT Nodes</th>
<th>Visits (max.)</th>
<th>Visits (avg.)</th>
<th>Time (avg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test (non-contr.) 12-attribute</td>
<td>4,605</td>
<td>2,301</td>
<td>926.8</td>
<td>0.3302</td>
</tr>
<tr>
<td>Test (cross-check) 12-attribute</td>
<td>8,097</td>
<td>2,788</td>
<td>1,216.9</td>
<td>0.4202</td>
</tr>
<tr>
<td>Test (cross-check) 73-attribute</td>
<td>8,097</td>
<td>2,857</td>
<td>1,632.5</td>
<td>0.6746</td>
</tr>
<tr>
<td>Test (simplified) 12-attribute</td>
<td>200</td>
<td>98</td>
<td>80.6</td>
<td>0.0228</td>
</tr>
<tr>
<td>Test (simplified) 73-attribute</td>
<td>200</td>
<td>98</td>
<td>81.6</td>
<td>0.0235</td>
</tr>
<tr>
<td>Content Server (cross-check)</td>
<td>36,593</td>
<td>8,705</td>
<td>2,331.9</td>
<td>0.5708</td>
</tr>
<tr>
<td>Content Server (simplified)</td>
<td>228</td>
<td>117</td>
<td>99.8</td>
<td>0.0247</td>
</tr>
</tbody>
</table>

6. Conclusion

We address the problem of discovering implicit redundancies in examples of communications, and using them to detect inconsistent values in incoming messages. This type of error can occur in circumstances that prevent the use of traditional error detection schemes, and detecting it is a challenging problem, not addressed in a general way by existing work. We have shown that the proposed approach can achieve a reasonable error detection coverage in fields where sequential relations exist, with the latency and false alarm rate being kept to a minimum by a heuristic that simplifies rule sets, favoring short rules that detect many errors over long rules with redundant information.

Besides proposing a general, receiver-based method for detecting inconsistent values, we highlight the fact that our approach discovers rules that are not found by existing work on the discovery of frequent episodes, considering, for example, messages with multiple attributes and types, and events involving variable assignments and comparisons. Note that the proposed approach may be more or less suited to specific protocols: it can only be used in cases where sequential relations exist, and can be described by the proposed definition for rules. With the proper adaptations, the results presented in this paper may also be relevant for other problems with similar characteristics, such as error detection in textual databases [12] or intrusion detection [6].

Future work includes proposing new mechanisms, able to discover more general rules, from noisy inputs or during operation time. Those improvements pose several new challenges, such as dealing with an infinite search space, distinguishing between noisy data and rare events, and the presence of faulty or even malicious systems.

References