Debugging Embedded Multimedia Application Traces through Periodic Pattern Mining

Patricia López Cueva[*†] Aurélie Bertaux[*] Alexandre Termier[*]

Jean François Méhaut[*] Miguel Santana[†]

[†]STMicroelectronics
Crolles, France
FirstName.LastName@st.com

[*]University of Grenoble, LIG
Grenoble, France
FirstName.LastName@imag.fr

ABSTRACT

Increasing complexity in both the software and the underlying hardware, and ever tighter time-to-market pressures are some of the key challenges faced when designing multimedia embedded systems. Optimizing the debugging phase can help to reduce development time significantly. A powerful approach used extensively during this phase is the analysis of execution traces. However, huge trace volumes make manual trace analysis unmanageable. In such situations, Data Mining can help by automatically discovering interesting *patterns* in large amounts of data. In this paper, we are interested in discovering periodic behaviors in multimedia applications. Therefore, we propose a new pattern mining approach for automatically discovering all periodic patterns occurring in a multimedia application execution trace.

Furthermore, gaps in the periodicity are of special interest since they can correspond to cracks or drop-outs in the stream. Existing periodic pattern definitions are too restrictive regarding the size of the gaps in the periodicity. So, in this paper, we specify a new definition of frequent periodic patterns that removes this limitation. Moreover, in order to simplify the analysis of the set of frequent periodic patterns we propose two complementary approaches: (a) a lossless representation that reduces the size of the set and facilitates its analysis, and (b) a tool to identify pairs of "competitors" where a pattern breaks the periodicity of another pattern. Several experiments were carried out on embedded video and audio decoding application traces, demonstrating that using these new patterns it is possible to identify abnormal behaviors.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging, Tracing; H.2.8 [**Database Management**]: Database Applications, Data Mining

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'12, October 7-12, 2012, Tampere, Finland.
Copyright 2012 ACM 978-1-4503-1425-1/12/09 ...$10.00.

Keywords

Embedded Systems, Multimedia Applications, Periodic Pattern Mining, Debugging

1. INTRODUCTION

The current trend in consumer electronics is to have products that support multimedia applications, such as set-top boxes, tablets, smartphones and MP4 players. All these products are powered by highly integrated System-on-a-Chip (SoC) solutions, that contain, in a single die, several processing units, memory blocks and specialized units for audio and video decoding. Developing efficient and robust applications for systems using these SoCs is a challenging issue. It is thus critical for companies developing embedded software to have comprehensive programming frameworks with advanced features for debugging and optimizing their applications.

Nowadays, the most widely used technique to give insight into the behavior of an embedded application for debugging and optimization purposes is *trace* analysis [19]. That is, to collect events generated by the system and the application in order to perform a post-mortem analysis of the execution. Many recent embedded systems directly integrate tracing hardware support in order to minimize intrusiveness, i.e. the act of tracing has minimal impact on the behavior of the application, allowing complex interactions to be shown in real-time applications such as video decoding.

Over the past decade, the software and hardware of embedded systems have faced an increase in complexity due to the use of techniques such as multi-threading, power and memory optimization, and so on. Consequently, the size of the execution traces of applications running on these systems has also increased, and we can only expect an even bigger increase with the introduction of many-core processors in embedded systems. Therefore, the manual analysis of execution traces is becoming an unmanageable task.

To overcome this issue we intend to use data mining in order to automatically extract pertinent information (called *patterns*) from traces. In this context, we consider a *pattern* as a set of functions and system events that are found together frequently in the trace. Multimedia applications have a periodic execution based on frame decoding, i.e. the same operations are performed on each frame or every n frames. Therefore, in this paper, we use *frequent periodic pattern*

mining on execution traces to discover periodic behaviors of multimedia applications.

However, sometimes this periodic behavior is not perfect: cracks in the sound or drop-outs in the video stream are often the consequence of a disruption in the periodicity of a pattern, showing up as a *gap*. Therefore, these *gaps* should be investigated as a priority for debugging and optimizing multimedia applications. The state of the art (see Section 7) considers only regular sized gaps, i.e. a multiple of the period, while in multimedia applications it is not possible to know the size of the gaps in advance. Therefore, we propose a new definition of periodic patterns that allows to have arbitrary sized gaps.

Frequent pattern mining generally generates a large amount of patterns which can make their analysis difficult. Therefore, we propose a reduced representation of the set of frequent periodic patterns, based on ternary relation theory [22] and the minimal generator concept [18], while keeping the same amount of information.

In this paper, we explore the current status of trace debugging on embedded systems as well as explain how to pre-process execution traces to analyze them with our pattern mining algorithm in Section 2. We propose a new definition of frequent periodic pattern in Section 3, and introduce our approach to reduce the set of outputted patterns mentioned above in Section 4. In Section 5, we present a new algorithm to mine the reduced set of frequent periodic patterns, as well as introduce an analysis tool that helps to identify pairs of competitor periodic patterns. By competitors we mean that their executions are in competition, i.e. a pattern breaking the periodicity of another pattern, which helps to identify possible conflicts between different parts of the system. In Section 6, we present the insights given by analyzing application traces. Next, we explore the state-of-the-art regarding pattern mining in system analysis, periodic pattern mining and pattern mining of ternary relations in Section 7. Finally, we present our conclusions and future work in Section 8.

2. MULTIMEDIA APPLICATIONS AND EMBEDDED PLATFORMS

In this section, we introduce the context in which trace debugging is used on embedded systems. We also explain how to pre-process the traces so that they can be analyzed by our pattern mining algorithm.

2.1 Trace Debugging on Embedded Systems

During the development of applications for embedded systems, development and evaluation boards are used by developers to test their applications. An example is the *STi7200-MBoard* platform [21] which contains an *STi7200* SoC, whose architecture is shown in Figure 1. This SoC is used in high-definition set-top boxes produced by *STMicroelectronics*. Apart from all the necessary connectivity, this system-on-chip contains a *ST40* core and two *ST231* cores. The *ST40* core is dedicated to application execution and device control, while the *ST231* cores are in charged of the audio and video decoding.

These development boards offer a way of collecting traces that can vary from a dedicated trace port, to a network connexion that allows the developer to retrieve the traces from the board. Execution traces provide a full view of the running system (operating system and application) while minimizing the intrusiveness.

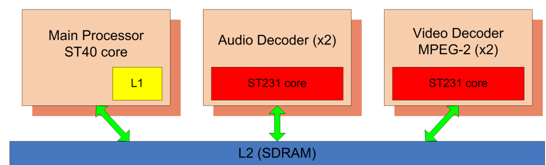

Figure 1: STi7200 SoC architecture

In some cases, software tracing solutions are provided by the operating system. For instance, on an *ST40* core, applications run on a Linux distribution for *STMicroelectronics* products. This operating system provides a tracing tool based on KProbes [8], which registers system and application events: interrupts, context switches, function calls, system calls, etc. In this paper, we focus on the analysis of software execution traces. Moreover, the execution traces used for the experiments were produced by the tracing tool mentioned above in this paragraph.

In other cases, hardware tracing solutions provide very accurate information about the execution of the system with no intrusion. The main problem with hardware-based tracing techniques is the extremely large amount of data generated. Moreover, transferring the trace out of the chip requires a large I/O bandwidth with a significant cost increase.

Embedded systems are complex entities composed of several processing units, accelerators, GPUs, DSPs and so on. Moreover, multimedia applications often consist of filter pipelines, in which certain steps are carried out by accelerators or GPUs instead of the main core. Therefore, in order to have an insight into what is really happening in the system, it is necessary to trace every component of the SoC. Companies such as Intel [16] or STMicroelectronics are currently working on system tracing solutions that tackle this necessity. Certainly traces obtained by this new approach will be a lot richer in terms of the amount of information, but also a lot bigger than current traces. In consequence, the need for automatic trace analysis tools will be even more important in the near future. Currently, we focus on the analysis of software execution traces, but our approach is generic and would be easily extended to be able to analyze this new generation of traces.

2.2 Execution Traces in Pattern Mining

An execution trace is a sequence of events and their timestamp, registered during the execution of an application, as shown in Figure 2. However, most pattern mining algorithms search for patterns common to a sequence of sets [1]. This makes it necessary to split the execution trace into sets of events, called *transactions*, in order to apply pattern mining algorithms.

We propose two methods to split the trace:

(a) using a time interval [11], where a transaction contains all events of the trace registered in the same time window, or

(b) using a function name, where a transaction contains all events of the trace that occurred between two occurrences of the given function.

As an example, we can observe the result of splitting a trace using a time interval of 0.1 ms in Figure 2. The timestamp of the events is not included in the transactions, so

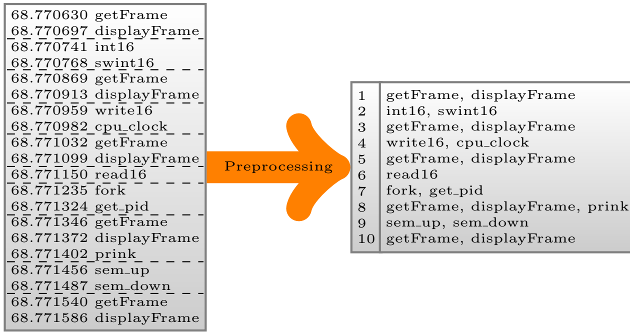

Figure 2: Preprocessing of an execution trace by splitting it into time intervals of 0.1 ms. Timestamp: seconds.microseconds

the information about when exactly an event was executed is not considered in the analysis. In this paper, we are interested in discovering sets of events that occur periodically in the execution trace, but the order in which those events are executed is not taken into account. Considering that the executing environment is multi-threaded, the order of the events might change according to the decision taken by the scheduler. Therefore, even if the order is not taken into account, our approach is able to discover interesting relationships between the events in the execution trace.

For a more exhaustive analysis where the order of event execution is considered important, there exist pattern mining techniques that can discover ordered sets of events, called sequences, that occur frequently in the trace. Nevertheless, these techniques are more complex and computationally expensive. The results obtained by these techniques might be interesting but these techniques are out of the scope of this paper.

Unless stated otherwise, in this paper we use the function that starts the decoding of a frame to split the trace in order to discover events that happen while decoding frames. Being able to identify the events that occur when decoding a frame allows us to more easily identify sets of events that might affect the application's behavior.

3. DEFINITIONS

In this section, we give basic definitions that allow us to present our definition of frequent periodic patterns.

As we have seen in Section 2, an execution trace is split into sets of events which, in data mining, are called *transactions*. We define our dataset $\mathcal{D}$ as the ordered set of transactions $\{t_1, t_2, ..., t_n\}$ obtained by splitting an execution trace, and the set of items $\mathcal{I}$ as the set of all possible events found in the trace. An itemset is a set of events belonging to $\mathcal{I}$, e.g. $\{sem_up,\ int16,\ cpu_clock\}$. For instance, Table 1 presents the dataset obtained after preprocessing the execution trace shown in Figure 2 by splitting the trace using a time interval of 0.1 ms.

DEFINITION 1. *Given a set of items $\mathcal{I} = \{i_1, i_2, ..., i_r\}$, a* ***dataset*** *$\mathcal{D}$ is an ordered set of transactions $\{t_1, t_2, ..., t_n\}$ where each transaction is a subset of $\mathcal{I}$, i.e. $t_k \subseteq \mathcal{I}$ for $1 \le k \le n$, and where the order is defined by $t_i < t_j$ if and only if $i < j$. The length of the dataset $\mathcal{D}$ is the number of transactions that form part of the dataset and is denoted by $|\mathcal{D}|$.*

Table 1: A dataset in the context of system trace analysis.

t_k	Itemset
t_1	getFrame, displayFrame
t_2	int16, swint16
t_3	getFrame, displayFrame
t_4	write16, cpu_clock
t_5	getFrame, displayFrame
t_6	read16
t_7	fork, get_pid
t_8	getFrame, displayFrame, printk
t_9	sem_up, sem_down
t_{10}	getFrame, displayFrame

An itemset X is denoted by $\{x_1, x_2, ..., x_j\}$ where x_t is an item, i.e. $x_t \in \mathcal{I}$. Considering $X \subseteq I$, we say that an itemset X *occurs* in the transaction t_k if and only if $X \subseteq t_k$.

Given a transaction t_k, its transaction identifier, denoted as $tid(t_k)$, is its position in the dataset, i.e. k. We also define the distance d between two transactions t_i and t_j as the difference between their transaction identifiers, i.e. $d = j - i$ with $i \le j$.

When an itemset occurs over a set of transactions and the distance between any two consecutive transactions is constant, this set of transactions forms a *cycle*.

DEFINITION 2. *Given an itemset X and a period p, a cycle of X, denoted* $\textbf{cycle}(o, p, l, X)$*, is a maximal set of l transactions in $\mathcal{D}$ containing X, starting at transaction t_o and separated by equal distance p: $cycle(o, p, l, X) = \{t_k \in \mathcal{D} | X \subseteq t_k, k = o + p * i, 0 \le i < l, X \nsubseteq t_{o-p}, X \nsubseteq t_{o+p*l}\}$ where $0 \le o < n$, $2 \le l \le n$ and n is the number of transactions in $\mathcal{D}$.*

EXAMPLE 1. *In the dataset in Table 1, the itemset $X = \{getFrame,\ displayFrame\}$ is found at a period $p = 2$ on transactions from t_1 $(o = 1)$ to t_5, therefore it forms a cycle of length $l = 3$, denoted $cycle(1, \mathbf{2}, 3, \{getFrame,\ displayFrame\}) = \{t_1, t_3, t_5\}$*[1].

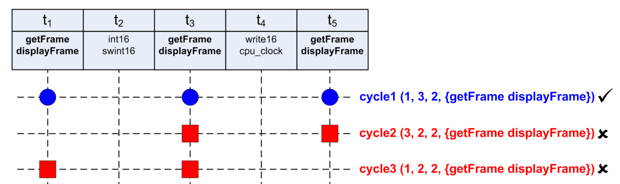

Figure 3: Examples of cycles and non-cycles

Figure 3 offers a graphical representation of a set of transactions where $cycle1(1, \mathbf{2}, 3, \{getFrame,\ displayFrame\})$ is maximal. $cycle2(3, \mathbf{2}, 2, \{getFrame,\ displayFrame\})$ and $cycle3(1, \mathbf{2}, 2, \{getFrame,\ displayFrame\})$ are not maximal since they can be extended with transactions t_1 and t_5 respectively, and therefore they are not valid cycles in our context.

A set of consecutive cycles (the end of a cycle happens before the beginning of the following cycle) over the same itemset and the same period forms a *periodic pattern*.

[1]Periods are presented in bold for clarity.

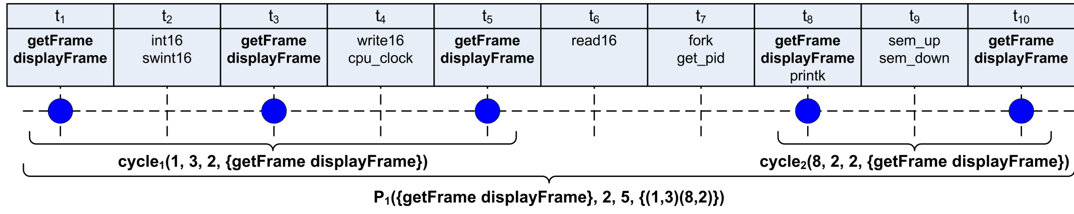

Figure 4: Periodic pattern formation

Definition 3. *An itemset X together with a set of cycles C and a period p form a **periodic pattern** if the set of cycles $C = \{(o_1, l_1), ..., (o_k, l_k)\}$[2], with $1 \leq k \leq m$ and m being the maximum number of cycles of period p in the dataset $\mathcal{D}$, is a set of cycles of X such that:*

1. *All cycles have the same period p.*
2. *All cycles are consecutive: $\forall (o_i, l_i), (o_j, l_j) \in C$ such that $1 \leq i < j \leq k$, we have $o_i < o_j$.*
3. *Cycles do not overlap: $\forall (o_i, l_i), (o_j, l_j) \in C$ such that $i < j$, we have $o_i + (p * (l_i - 1)) < o_j$.*

We denote this periodic pattern $P(X, p, s, C)$.

The **support** of a periodic pattern, denoted s, is the sum of all cycle lengths in C, i.e. given $C = \{(o_1, l_1), ..., (o_k, l_k)\}$ with $1 \leq k \leq m$ then $s = \sum_{i=1}^{k} l_i$.

Example 2. *In Figure 4 we can observe that the cycles $cycle_1(1, \mathbf{2}, 3, \{getFrame,\ displayFrame\})$ and $cycle_2(8, \mathbf{2}, 2, \{getFrame,\ displayFrame\})$ form a periodic pattern $P_1 = (\{getFrame,\ displayFrame\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$ with period 2 and a support of 5 transactions.*

We now introduce the notion of *frequent periodic patterns.*

Definition 4. *Given a minimum support threshold min_sup, a periodic pattern P is **frequent** if its support is greater than min_sup, i.e. $P(X, p, s, C)$ is frequent if and only if $s \geq min_sup$.*

Example 3. *Given the dataset shown in Table 1 and a minimum support of two transactions, the set of frequent periodic patterns is presented in Table 2*[3].

As we can see in Table 2, the set of frequent periodic patterns is highly redundant. On one hand, all combinations of large itemsets are consider as patterns. For example, P_1, P_2 and P_3 are present in exactly the same transactions and the itemset of P_3 contains the itemsets of P_1 and P_2. Therefore P_1 and P_2 do not give any more information than P_3. On the other hand, combinations of small periods by addition or multiplication generate redundant patterns. For example, P_9 is redundant with respect to P_3 since its period is multiple of P_3's period while the transactions in P_9 can be found in P_3. Therefore P_9 does not give any more information than P_3.

In real datasets tens of thousands of frequent periodic patterns might be found, which are impractical to analyze by an application developer. In order to produce a reduced representation of the set of frequent periodic patterns we adopt a triadic approach by introducing the periods into the dataset.

[2] For simplicity of notation, a cycle is represented here by its origin o and its length l since all cycles in the set share the same itemset X and period p.

[3] Only periods not greater that the length of the dataset divided by min_sup are considered.

Table 2: Set of frequent periodic patterns

Frequent Periodic Patterns
$P_1(\{getFrame\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$
$P_2(\{displayFrame\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$
$P_3(\{getFrame,\ displayFrame\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$
$P_4(\{getFrame\}, \mathbf{3}, 2, \{(5,2)\})$
$P_5(\{displayFrame\}, \mathbf{3}, 2, \{(5,2)\})$
$P_6(\{getFrame,\ displayFrame\}, \mathbf{3}, 2, \{(5,2)\})$
$P_7(\{getFrame\}, \mathbf{4}, 2, \{(1,2)\})$
$P_8(\{displayFrame\}, \mathbf{4}, 2, \{(1,2)\})$
$P_9(\{getFrame,\ displayFrame\}, \mathbf{4}, 2, \{(1,2)\})$
$P_{10}(\{getFrame\}, \mathbf{5}, 2, \{(3,2)\})$
$P_{11}(\{displayFrame\}, \mathbf{5}, 2, \{(3,2)\})$
$P_{12}(\{getFrame,\ diaplayFrame\}, \mathbf{5}, 2, \{(3,2)\})$
$P_{13}(\{getFrame\}, \mathbf{5}, 2, \{(5,2)\})$
$P_{14}(\{displayFrame\}, \mathbf{5}, 2, \{(5,2)\})$
$P_{15}(\{getFrame,\ displayFrame\}, \mathbf{5}, 2, \{(5,2)\})$

4. NON-REDUNDANT PATTERNS

In this section, we present the application of ternary relation theory to periodic pattern mining and the definition of minimal periodic generators that will allow us to generate a reduced set of periodic patterns that are easier to analyze by the developer.

As we have seen in Section 3, the set of frequent periodic patterns is highly redundant. This is a common drawback when using pattern mining techniques which complicates the analysis of the results. A major breakthrough has been the study of closed operators applied to patterns such as itemsets [18], gradual patterns [2] and so on. The set of closed patterns, mined by closed operators, is a reduced representation of the set of frequent patterns but contains the same information.

No previous study has been carried out regarding closed periodic patterns. This might be because the theory behind closed patterns, called Galois connection, is based on binary relations, such as the relation $items \times transactions$ for itemset mining. Analyzing the structure of periodic patterns, we can observe that they are based on ternary relations involving the items, the transactions and the periods. In ternary relations, it is not possible to make use of a Galois connection to generate closed patterns [3], therefore it is not possible to apply the Galois theory to periodic pattern mining.

Instead, we are going to use a triadic approach to Formal Concept Analysis [10] to mine a reduced representation of the set of frequent periodic patterns, which in this context is called the set of **triadic concepts**. Nevertheless, since all possible periods are considered, the set of triadic concepts might contain redundant periods, such as multiples of smaller periods, periods formed by the concatenation of two smaller periods, and so on. So, to reduce this redundancy, we define the concept of minimal periodic generators which allows us to extract the minimal set of periodic patterns

Table 3: Representation of the relation $\mathcal{Y}$

$I/P-T$	2										3									
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
getFrame	×		×		×			×		×					×			×		
displayFrame	×		×		×			×		×					×			×		
...																				

$I/P-T$	4										5									
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
getFrame	×				×								×		×			×		×
displayFrame	×				×								×		×			×		×
...																				

that explains the behavior of the application, without losing information.

4.1 Triadic Approach to Periodic Pattern Mining

Our dataset, introduced in Definition 1, corresponds to a relation between two attributes, *items* × *transactions*, i.e. $\mathcal{R} \subseteq \mathcal{I} \times \mathcal{D}$ and each $r \in \mathcal{R}$ can be represented by a couple $r = \{(i,t)|i \in \mathcal{I}, t \in \mathcal{D}\}$, denoting that the item i occurs on transaction t.

In order to mine periodic patterns, the period should be included in the dataset, but in order to do so, the binary relation $\mathcal{R} \subseteq \mathcal{I} \times \mathcal{D}$ has to be transformed into a ternary relation $\mathcal{Y} \subseteq \mathcal{I} \times \mathcal{P} \times \mathcal{D}$, with $\mathcal{P}$ the set of all possible periods that we limit to the range $[1..|\mathcal{D}|/min_sup]$.

To formalize this ternary relation we are going to introduce a triadic approach to formal concept analysis first introduced by Lehmann et al. [10] in 1995. Specifically, the concepts of triadic context and triadic concepts are presented here including some modifications needed in order to adapt them to periodic pattern mining.

Definition 5. *A* ***periodic triadic context*** *is defined as a quadruple $(\mathcal{I}, \mathcal{P}, \mathcal{D}, \mathcal{Y})$ where $\mathcal{I}$ is the set of items, $\mathcal{P}$ is the set of periods, $\mathcal{D}$ is the set of transactions, and $\mathcal{Y}$ is a ternary relation between $\mathcal{I}$, $\mathcal{P}$ and $\mathcal{D}$, i.e. $\mathcal{Y} \subseteq \mathcal{I} \times \mathcal{P} \times \mathcal{D}$.*

An element of the relation $y \subseteq \mathcal{Y}$ is denoted by the triple $y = \{(i,p,t)|i \in \mathcal{I}, p \in \mathcal{P}, t \in \mathcal{D}\}$ and is read: the transaction t forms part of a cycle of period p of the item i.

Example 4. *Given the dataset shown in Table 1, the corresponding triadic context is shown in Table 3*[4]*. Each cross in the table represents an element of the ternary relation $\mathcal{Y}$. For example, $P_1(\{getFrame\}, 2, 5, \{(1,3)(8,2\})$ in Table 2 is transformed into the triples $(getFrame, 2, t_1)$, $(getFrame, 2, t_3)$, $(getFrame, 2, t_5)$, $(getFrame, 2, t_8)$, $(getFrame, 2, t_{10})$ shown by the corresponding crosses in Table 3.*

Definition 6. *Given a minimum support threshold min_sup, a triple (I, P, T), with $I \subseteq \mathcal{I}$, $P \subseteq \mathcal{P}$, $T \subseteq \mathcal{D}$ and $I \times P \times T \subseteq \mathcal{Y}$, is* ***frequent*** *if and only if $I \neq \emptyset$, $P \neq \emptyset$ and $|T| \geq min_sup$.*

Example 5. *In Table 3, given a min_sup of 2, we can observe several frequent triples such as $(\{getFrame\}, \{2,3\}, \{t_5, t_8\})$ or $(\{getFrame,\ displayFrame\}, \{5\}, \{t_3, t_5, t_8, t_{10}\})$, since the number of transactions forming those triples is greater or equal to 2.*

[4]All items, excluding $getFrame$ and $displayFrame$, do not form any cycle of any possible period and, for clarity, they are not included in the table.

The set of frequent triples is highly redundant since it includes all possible combinations between items, periods and transactions included in the ternary relation $\mathcal{Y}$. A *lossless* reduced representation of this set was introduced by Wille [22] and named *triadic concepts*. Saying the representation is lossless means that it is possible to reconstruct the set of frequent triples from the set of triadic concepts without any extra information.

Definition 7. *A* ***triadic concept*** *of a triadic context $(\mathcal{I}, \mathcal{P}, \mathcal{D}, \mathcal{Y})$ is a triple (I, P, T) with $I \subseteq \mathcal{I}$, $P \subseteq \mathcal{P}$ and $T \subseteq \mathcal{D}$, such that none of its three components can be enlarged without violating the condition $I \times P \times T \subseteq \mathcal{Y}$.*

Example 6. *In Table 4, we can observe the set of triadic concepts extracted from the set of frequent triples obtained from the dataset shown in Table 3. The triples forming this set are triadic concepts since it is not possible to extend any of the attributes of the triple without violating the relation $\mathcal{Y}$.*

Table 4: Set of triadic concepts

Triadic Concepts
$T_1(\{getFrame,\ displayFrame\}, \{2\}, \{t_1, t_3, t_5, t_8, t_{10}\})$
$T_2(\{getFrame,\ displayFrame\}, \{2,4\}, \{t_1, t_5\})$
$T_3(\{getFrame,\ displayFrame\}, \{2,5\}, \{t_3, t_5, t_8, t_{10}\})$
$T_4(\{getFrame,\ displayFrame\}, \{2,3,5\}, \{t_5, t_8\})$
$T_5(\{getFrame,\ displayFrame\}, \{2,3,4,5\}, \{t_5\})$

It can be observed that the set of triadic concepts is a lossless representation of the set of frequent periodic patterns even if they are presented using a different notation. Moreover, it is important to note that the set of triadic concepts presented on Table 4 is considerably smaller than the set of frequent periodic patterns presented in Table 2.

This set can be translated into a set of frequent periodic patterns by calculating the cycles included in the set of transactions of each triadic concept. For instance, $T_2(\{getFrame,\ displayFrame\}, \{2,4\}, \{t_1, t_5\})$ contains only one cycle of period 4 with transactions t_1 and t_5 which would give us the periodic pattern $(\{getFrame,\ displayFrame\}, 4, 2, \{(1,2)\})$ (period=4, support=2, cycle offset=1 and cycle length=2) which is P_9 from Table 2, being able to deduce as well P_7 and P_8 since their itemsets are subsets of $\{getFrame,\ displayFrame\}$.

Nevertheless, this set still contains redundant information in terms of redundant periods. For example, if we consider the triadic concepts T_1 and T_2 from the set of triadic concepts shown in Table 4, we can see that T_2 is "included" in T_1, i.e. they have the same itemset and the transactions belonging to T_2 are a subset of the transactions belonging to T_1 and therefore, period 4 of T_2 can be "deduced" from the set of transactions of T_1. As a result, T_2 can be removed

without losing information. The same logic can be applied to T_3, T_4 and T_5, reducing the set to pattern T_1.

The way of deducing T_2 from T_1 is by calculating the possible periods between all transactions in T_1, and then generating the cycles belonging to each period, this way, from T_1 we can obtain periods 2 ($\{t_1,t_3,t_5,t_8,t_{10}\}$), 3 ($\{t_5,t_8\}$), 4 ($\{t_1,t_5\}$) and 5 ($\{t_3,t_5,t_8,t_{10}\}$). The last step is to calculate the maximal subsets involving several periods on the same set of transactions as is the case with t_1 and t_5 which are found in periods 2 and 4 generating the triadic concept ($\{getFrame,\ displayFrame\}, \{2, 4\}, \{t_1, t_5\}$).

In order to obtain a reduced representation of the set of triadic concepts we propose removing triadic concepts with redundant periods. For this, we present here the definition of *minimal periodic generator* which allows us to extract the set of triadic concepts that does not contain redundant periods.

4.2 Minimal Periodic Generators

In general terms, a minimal generator is the smallest pattern that will determine a closed pattern using the closure operator [18]. In this context, a minimal periodic generator is the triadic concept with the smallest set of periods and the biggest set of transactions that can determine another triadic concept using the method introduced above. For it to determine another triadic concept they have to share the same itemset, it has to include all transactions of the other triadic concept and have a smaller set of periods than the other triadic concept. Indeed, the set of periods of the minimal periodic generator is a subset of the set of periods of the other triadic concept. This is because the set of transactions of the triadic concept are included in the set of transactions of the minimal periodic generator, and therefore all those transactions have associated the periods of the minimal periodic generator and a few more.

DEFINITION 8. *A triadic concept (I, P, T) is a **minimal periodic generator** if there does not exist any other triadic concept (I', P', T') such that $I = I'$, $P' \subset P$ and $T' \supset T$.*

EXAMPLE 7. *In Table 5, we can observe the set of minimal periodic generators extracted from the set of triadic concepts shown in Table 4. For instance, $T_2(\{getFrame, displayFrame\}, \{2, 4\}, \{t_1, t_5\})$ is not a minimal periodic generator since there exists $T_1(\{getFrame,\ displayFrame\}, \{2\}, \{t_1, t_3, t_5, t_8, t_{10}\})$ with the same itemset $\{getFrame, displayFrame\}$, a smaller set of periods $\{2\} \subset \{2, 4\}$ and a bigger set of transactions $\{t_1, t_3, t_5, t_8, t_{10}\} \supset \{t_1, t_5\}$.*

Table 5: Set of minimal periodic generator

Minimal Periodic Generators
$M_1(\{getFrame,\ displayFrame\}, \{2\}, \{t_1, t_3, t_5, t_8, t_{10}\})$

It is important to note that the set of minimal periodic generators shown in Table 5 is considerably smaller than the set of triadic concepts shown in Table 4, and therefore smaller than the set of frequent periodic patterns shown in Table 2, and that it does not contain redundant periods.

In order to fit the periodic pattern notation introduced in Definition 3, the set of minimal periodic generators should be post-processed. For each minimal periodic generator the set of periods is extracted. Then, for each period in the set of periods, the set of cycles corresponding to that period is generated by reading the set of transactions, generating a new periodic pattern containing the period and the set of cycles.

In the example, from Table 5 which contains only one minimal periodic generator $M_1(\{getFrame,\ displayFrame\}, \{2\}, \{t_1,t_3,t_5,t_8,t_{10}\})$, we obtain the frequent periodic pattern $P(\{getFrame,\ displayFrame\}, 2, 5, \{(1, 3)(8, 2)\})$, which corresponds to P_3 from Table 2. This is a reduced representation of the set of frequent periodic patterns containing enough information to deduce all other frequent periodic patterns. As explained in the previous section, the set of triadic concepts can be deduced from the set of minimal periodic generators. Then, the whole set of frequent periodic patterns can be deduced from the set of triadic concepts by generating all triples contained in the set and then grouping them by item and period (taking into account all possible combinations between the items to form the itemsets).

5. MINING PERIODIC PATTERNS

In this section, we introduce an algorithm to mine minimal periodic generators from a dataset, called *PerMiner*. We also present a tool called *Competitors Finder* that helps to identify pairs of "competitors", pairs of periodic patterns where a pattern breaks the periodicity of another pattern. This tool helps to identify possible conflicts between different entities in the system.

Algorithm 1 Periodic Pattern Miner

```
1: procedure PerMiner(I, D, min_sup)
Input: Itemset I, dataset D, minimum support min_sup
Output: All frequent periodic patterns that occur in D
2: TS := TripleMiner(I, D, min_sup)
3: TC := DATA-PEELER (TS, min_sup)
4: MPG := MPGMiner(TC)
5: FPP := PostProcess(MPG)
6: Print(FPP)
7: end procedure
```

The *PerMiner* algorithm in Algorithm 1, is divided into three steps:

1. The set of triples TS is generated. All possible periods with their transactions are calculated for individual items and then outputted in the form of triples.

2. The set of triadic concepts TC is generated from the set of triples TS. For this step an existing algorithm called DATA-PEELER [3] has been used.

3. The set of minimal periodic generators MPG is generated from the set of triadic concepts TC.

The set of minimal periodic generators MPG set is then transformed into frequent periodic patterns before being outputted.

The procedure $TripleMiner$ in Algorithm 2 generates all triples (i, p, t), i.e. for each item and for each possible period in the range $[1..|\mathcal{D}|/min_sup]$ it searches all cycles of the selected period with transactions containing the selected item. The objective of this function is to transform our dataset into a triadic context exploitable by DATA-PEELER.

The function *build_triples* in Algorithm 3 is used by *TripleMiner* to generate all triples of a given item and a given period. For this, it scans the support set of the item i, given

Algorithm 2 Triple Miner

```
 1: procedure TripleMiner(I, D, min_sup)
Input: Itemset I, dataset D, minimum support min_sup
Output: All triples that occur in D
 2: TS ⇐ ∅
 3: for all i ∈ I do
 4:     for all p ∈ [1, D.size/min_sup] do
 5:         TS := TS ∪ build_triples(i, p, D)
 6:     end for
 7: end for
 8: return TS
 9: end procedure
```

by $D[\{i\}]$, which is the set of transactions of D in which i occurs. For each transaction in $D[\{i\}]$, the function tries to build a cycle of the given period starting in that transaction. If a cycle is found, its length being at least two transactions, the cycle is transformed in triples and added to the output list. Visited transactions are marked to avoid generating redundant cycles.

Algorithm 3 Build triples

```
 1: function build_triples(i, p, D)
Input: Item i, period p, dataset D
Output: Set of triples of item i and period p
 2: L_T ← ∅
 3: for all t ∈ D[{i}] do
 4:     if t not visited yet then
 5:         t ← visited; len ← 0; next ← tid(t)
 6:         while t_next ∈ D[{i}] do
 7:             t_next ← visited; len := len + 1; next := next + p
 8:         end while
 9:         if len > 2 then
10:             while len > 0 do
11:                 L_T := L_T ∪ (i, p, tid(t) + ((len − 1) ∗ p))
12:             end while
13:         end if
14:     end if
15: end for
16: return L_T
17: end function
```

DATA-PEELER generates all triadic concepts, containing at least *min_sup* transactions, contained in the set of triples generated by *TripleMiner*. Then, the procedure *MPG-Miner* in Algorithm 4 is in charge of generating the set of minimal periodic generators from the set of triadic concepts generated in the previous step. Following Definition 8, the function compares the set of triadic concepts two by two, and if it finds a triadic concept that is "included" in another triadic concept in the set, the former is deleted from the list. As explained in section 4.2, by "included" we mean that they have the same itemset, the set of periods of the latter is included in the set of periods of the former, and the set of transactions of the former is included in the set of periods of the latter.

The procedure *PostProcessMPG* in Algorithm 5 generates a set of frequent periodic patterns from the set of minimal periodic generators MPG. For each period p in the set of periods P of each minimal periodic generator (I, P, T), *build_cycles* generates all possible cycles of period p that can be formed using the set of transactions T of the minimal periodic generator.

Algorithm 4 Minimal Periodic Generators Miner

```
 1: procedure MPGMiner(CTS)
Input: Set of triadic concepts TC
Output: Set of minimal periodic generators MPG
 2: MPG ⇐ TC
 3: for all (I, P, T) ∈ MPG do
 4:     for all (I′, P′, T′) ∈ MPG
              with (I, P, T) ≠ (I′, P′, T′) do
 5:         if I == I′ AND P ⊂ P′ AND T ⊃ T′ then
 6:             MPG := MPG \ (I′, P′, T′)
 7:         end if
 8:     end for
 9: end for
10: return MPG
11: end function
```

Algorithm 5 Post processing of the set of Minimal Periodic Generators

```
 1: procedure PostProcessMPG(MPG)
Input: Set of minimal periodic generators MPG
Output: Set of frequent periodic patterns FPP
 2: FPP ⇒ ∅
 3: for all (I, P, T) ∈ MPG do
 4:     for all p ∈ P do
 5:         FPP.add(I, p, build_cycles(p, T))
 6:     end for
 7: end for
 8: return FPP
 9: end function
```

The function *add* includes the itemset I, with the period p and the associated cycles to the list of frequent periodic patterns in the form of frequent periodic pattern. If there exists a frequent periodic pattern in the set with the same itemset and the same period, the sets of cycles are joined, checking if there exist overlap between any two cycles of the set. If an overlap between two cycles is found, the frequent periodic pattern is split in as many patterns as needed in order not to have overlaps between the cycles of the same set. Functions *build_cycles* and *add* are not detailed here due to lack of space, but their implementation is straightforward.

5.1 Competitors Finder

Here we introduce a new analysis tool called *Competitors Finder* that helps to identify pairs of competitor periodic patterns from the set of frequent periodic patterns deduced from the set of minimal periodic generators. This tool allows the developer to easily identify possible conflicts between different parts of the system, i.e. between the application and the operating system, between different modules or drivers of the operating system, and so on, saving a significant amount of time to the developer. In this sense, we consider that a conflict or competition between two patterns is simply the inverse of the overlap between the two patterns. This is, if one pattern P_1 disrupts another pattern P_2, then during the disruption P_1 will be active but not P_2. So, our objective is to automatically identify these situations.

The procedure *CompetitorsFinder* in Algorithm 6 receives the set of frequent periodic patterns deduced from

Algorithm 6 Competitors Finder

1: **procedure** $CompetitorsFinder(MPG', min_ratio)$
Input: Set of frequent periodic patterns MPG', minimum rate of competition min_ratio
Output: Set of pairs of competitors $MaxComp$
2: $MaxComp \Leftarrow \emptyset$
3: **for all** $p \in MPG'$ **do**
4: $max_ratio := 0$
5: **for all** $p' \in MPG'$ with $p \neq p'$ **do**
6: $comp_ratio := competition_ratio(p, p')$
7: **if** $comp_ratio > max_ratio$ **then**
8: $max_ratio \Leftarrow comp_ratio$; $max_pat \Leftarrow p'$
9: **end if**
10: **end for**
11: **if** $(max_pat, p, *) \notin MaxComp$ AND $max_ratio \geq min_ratio$ **then**
12: $MaxComp.add(p, max_pat, max_ratio)$
13: **end if**
14: **end for**
15: **return** $MaxComp$
16: **end procedure**

the set of minimal periodic generators, and for each frequent periodic pattern it searches the pattern that is the most in competition with it. For each pattern, this procedure compares it with all other patterns and calculates the competition ratio by invoking *competition_ratio* function. In *max_ratio* and *max_pat*, the procedure stores the maximum ratio found and the pattern linked to that ratio.

When the pattern has been compared to all other patterns, the procedure checks if the competition ratio is bigger than the competition ratio threshold given as an input, and as well, whether the two patterns were already stated as competitors, in which case they are not outputted to avoid repetitions. Finally the list of competitors with their competition ratio is returned by this procedure.

Algorithm 7 Competition Ratio Calculator

1: **function** $competition_ratio(P, P')$
Input: Two periodic patterns
Output: The ratio of competition between P and P'
2: **for all** $c_i, c_{i+1} \in P.cycles$ **do**
3: **for all** $c' \in P'.cycles$ **do**
4: $match \mathrel{+}= calculate_coexecution(c, c')$
5: $match \mathrel{+}= calculate_cogap(c_i, c_{i+1}, c')$
6: **end for**
7: **end for**
8: **return** $(1 - ((num_trans - match)/num_trans)) * 100;$
9: **end function**

The function *competition_ratio* in Algorithm 7 calculates the competition ratio between two frequent periodic patterns. By comparing the cycles between them, it uses the function *calculate_coexecution* to calculate the area of co-execution of two cycles, and the function *calculate_cogap* to calculate the area between two cycles of the first periodic patterns (gap) not occupied by a cycle of the second periodic pattern. The sum of all the output values of these two functions is called matching ratio, and the competition ratio is the result of calculating just the opposite, i.e. 100% - matching ratio.

6. EXPERIMENTAL RESULTS

In this section, we present several experiments carried out on embedded multimedia application traces. The experiments show that periodic pattern mining can help to debug applications by automatically extracting representative information from their execution traces. These experiments could have been carried out by analyzing exclusively the set of frequent periodic patterns but it would have taken a lot longer since the difference in size between the set of frequent periodic patterns and the set of minimal periodic generators is considerable as stated below.

Figures 5 and 6 correspond to two visualization tools developed to facilitate the analysis of the periodic patterns. In Figure 6 the list of periodic patterns can be explored by navigating the list of itemsets. When the user selects an itemset from the list, the periodic patterns associated with that itemset, one for each period, are shown on the bottom part of the tool. Each line corresponds to a periodic pattern with the period value on the left part of the figure and the occurrences on the right. The occurrences are visualized by a sequence of vertical lines representing all possible transactions on the dataset (from left to right). A line is colored when the corresponding transaction forms part of the selected periodic pattern. Similarly, Figure 5 shows the list of pairs of competitors. When a pair of competitors is selected, the occurrences of both patterns are shown, pattern 1 on top and pattern 2 at the bottom.

6.1 HNDTest Application

In this example, we retrieved a trace from an execution of a video and audio decoding test application called *HNDTest*, test application for *STMicroelectronics* development boards, on an *ST40* processor, introduced in Section 2.1. Then, we preprocessed it, as explained in Section 2.2, in order to split the trace into frames. The execution trace occupied 7.2 MB of memory. After preprocessing, the dataset contained 240 frames, and with a support threshold of 10% our algorithm mined 859 minimal periodic generators, faster to analyze than 7.2 MB of execution trace. This trace would produce 109,668 triadic concepts and 38,459 frequent periodic patterns. Therefore, we can observe that mining minimal periodic generators considerably reduces the number of patterns to analyze.

The bigger number of triadic concepts with respect to the number of frequent periodic patterns can be explained by the fact that the number of periods in frequent periodic patterns is limited to one per pattern while in the triadic concepts is not limited. Consequently, all possible combinations of all lengths of the set of possible periods are generated as triadic concepts. All these extra patterns are then removed by the minimal periodic generator miner. Also, the transactions of the dataset used in this example have an average of 8 items per transaction which produces patterns with relatively short itemsets that consequently do not produce many combinations in terms of frequent periodic patterns.

As part of the analysis of the set of frequent periodic patterns mined, we used the tool *Competitors Finder* (see Section 5.1) to identify possible conflicts between different entities of the system. We highlight here a pair of competitors found by the tool involving on one hand, *Interrupt 16*, which is the clock of the processor, and *Interrupt 168*, which is a USB port interrupt, and on the other hand, a context

Figure 5: Conflict between application and operating system

switch and a system call (`try_to_wake_up`) involving thread *HNDTest*.

In Figure 5 we can observe that these two patterns are in competition: the top pattern (interrupts) stops executing when the bottom pattern (*HNDTest*) increases its frequency. In this context, the processor periodically polls the device connected to the USB port to check whether it has data to transfer. This is achieved using interrupt 168. When *HNDTest* increases its activity, it masks the interrupts and therefore prevents the processor from transferring any incoming data from the USB port, which causes a delay in transmission. This might have further repercussions if the USB reception buffer becomes full while waiting to transfer the data, causing the data to be overwritten.

6.2 Gstreamer Application

GStreamer is a pipeline-based multimedia framework that has been adopted by many different corporations such as Nokia, Texas Instruments and so on. In this experiment, we used a trace that registered an audio decoding applications, using GStreamer, while playing back an audio file. The application was executed on a platform that contained an ARM processor over an Orly SoC [20]. Our algorithm, with a support threshold of 10%, mined 1,467 minimal periodic generators. This trace would have produced 21,588 triadic concepts and 3,086,321 frequent periodic patterns.

In this example, the number of frequent periodic patterns is much bigger than the number of triadic concepts. This is because the transactions in the dataset of this example are very long, 35 items per transaction in average, and therefore generate patterns with a long itemset. For this reason, the number of frequent periodic patterns per itemset and per period is large since there is a frequent periodic pattern for each combination (of any length) of the items in the itemset.

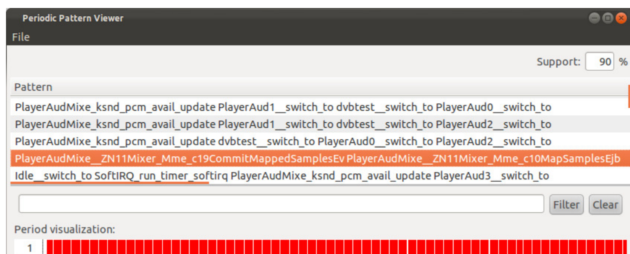

Figure 6: Trace visualization

In this context, the mixer maps audio samples every 32ms, so we decided to split the trace using a time interval of 32ms to see whether the expected period was preserved or not. In the minimal periodic generators mined, we expected to find some patterns with a period of 1 (32ms) over all the data. Surprisingly, the patterns found exhibit *gaps* in the periodicity (vertical white lines in period visualization in Figure 6) showing that the application is unable to keep up the expected mixing rate.

The application developers investigated these *gaps* and found that there was a bug in the calculation of an interrupt period. This interrupt was in charge of flushing a buffer of samples read from memory but it was being generated too late causing buffer overflows and higher level drivers to underflow when operating double buffered.

Discussion. In these experiments, we have shown that our approach allows developers to quickly discover certain problems in the execution of their applications by automatically analyzing their execution traces, that otherwise would have taken a long time to be discovered by manually analyzing execution traces.

7. RELATED WORK

Pattern mining is starting to play an important role in system analysis, even more in embedded systems, where tracing is widely used in order to analyze the system while avoiding high intrusiveness.

First uses of pattern mining in system analysis focused on detecting bugs, e.g. introduced by copying-and-pasting kernel source code [12] or caused by the violation of programing rules [13], or more generally detecting systemic problems [9]. Lo et al. [14] studied how to classify software behaviors, obtained by pattern mining of known normal and failing execution traces, in order to detect failures in future executions of the system.

Recently, Chang et al. [4] worked on system verification using pattern mining in order to extract assertions from simulated traces of the system being validated. In terms of performance analysis, Zou et al. [23] used pattern mining to reduce vast amounts of hardware sample data into a set of easier-to-analyze frequent instruction sequences, in order to help to analyze the performance of the system.

Nevertheless, none of the previous studies have applied periodic pattern mining to system analysis.

The first studies carried out on periodicity focused on association rules. As an example Ozden et al. [17] looked for cyclic association rules in transactional databases. Their objective was to find association rules that hold in all segments of the database over a given period. This kind of periodicity, called perfect periodicity, is very useful for certain contexts but is too restrictive in our case.

Han et al. [6, 5] introduced a certain confidence in the periodicity which means that the pattern does not need to be found in all instances, but allows certain misses. Nevertheless, when there is a gap in the periodicity of the pattern, this gap is always regular, i.e. multiple of the period.

The first study to include irregularity in the periodicity was carried out by Ma et al. [15]. In their study, the authors extended the concept of partial periodicity introduced by Han et al. considering that periodic behavior might be found in part of the dataset and it is not necessarily expected to be persistent. The authors allowed gaps in the periodicity but they restricted their sizes to a certain time window.

None of the previous studies regarding periodic pattern mining have allowed for irregular gaps without any restriction on the length of the gap. It is important for us to allow irregularities in the gaps in order to discover different types of periodic patterns, e.g. patterns that are periodic during certain segments of the trace with big gaps between those segments, or patterns with a regular periodicity but that every time there is a gap the pattern gets shifted by n positions.

Triadic Concept Analysis was first introduced by Rudolf Wille in 1995 [22, 10]. Since then, several algorithms that mine triadic concepts have been proposed, among them we can find CubeMiner proposed by Ji et al. [12], Trias proposed by Jaschke et al. [7] and Data-Peeler proposed by Cerf et al. [3]. The latter was chosen in this paper since the authors showed through an experimental comparison that their algorithm was more efficient than CubeMiner or Trias. Moreover, the authors generalized the computation of triples previously studied by [12] and [7] to a constraint-based approach for mining closed sets from n-ary relations. But none of the previous studies have applied ternary relation theory to mining periodic patterns.

8. CONCLUSIONS AND FUTURE WORK

In the context of analyzing traces of multimedia embedded applications, we have presented a new definition of periodic patterns that allows unrestricted-sized gaps in the periodicity. Moreover, we have presented a lossless representation of the set of frequent periodic patterns, called minimal periodic generators, that simplifies the analysis. Also, we have introduced an algorithm for mining such minimal periodic generators and a tool that identifies pairs of competitors from the set of minimal periodic generators.

Through experiments on execution traces, we have demonstrated how pattern mining can help to discover problems more quickly than manually analyzing raw execution traces. We have applied our mining method and our *Competitors Finder* tool to an execution trace and identified a conflict between the application and the operating system, demonstrating the interest of our approach.

We have also shown how discovering the periodic behavior of multimedia applications can help to pinpoint when this periodicity is lost and therefore what affects the expected execution of the application.

The scope of this paper is to present our approach, its applicability and its interest. We intent to study the use of other patterns such as sequences (ordered itemsets) which would help to discover other behaviors of the application, and therefore complete the analysis. Moreover, we are planning to design an algorithm to mine minimal periodic generators directly, without having to generate the set of triadic concepts, in order to increase efficiency.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, San Francisco, CA, USA, 1994.

[2] S. Ayouni, A. Laurent, S. B. Yahia, and P. Poncelet. Mining closed gradual patterns. In *ICAISC*, pages 267–274, 2010.

[3] L. Cerf, J. Besson, C. Robardet, and J.-F. Boulicaut. Closed patterns meet n-ary relations. *TKDD*, 2009.

[4] P.-H. Chang and L.-C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *ASPDAC*, pages 607–612, 2010.

[5] J. Han. Efficient mining of partial periodic patterns in time series database. In *ICDE*, pages 106–115, 1999.

[6] J. Han, W. Gong, and Y. Yin. Mining segment-wise periodic patterns in time-related databases. In *KDD*, pages 214–218, 1998.

[7] R. Jaschke, A. Hotho, C. Schmitz, B. Ganter, and G. Stumme. TRIAS An algorithm for mining iceberg tri-lattices. In *ICDM*, pages 907–911, 2006.

[8] R. Krishnakumar. Kernel korner: KProbes-A kernel debugger. *Linux J.*, 2005.

[9] C. LaRosa, L. Xiong, and K. Mandelberg. Frequent pattern mining for kernel trace data. In *SAC*, pages 880–885, 2008.

[10] F. Lehmann and R. Wille. A triadic approach to formal concept analysis. In *Conceptual Structures: Applications, Implementation and Theory*, volume 954 of *Lecture Notes in Computer Science*, pages 32–43. Springer Berlin / Heidelberg, 1995.

[11] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-Miner: Mining block correlations in storage systems. In *FAST*, pages 173–186, 2004.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *TSE*, pages 176–192, 2006.

[13] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, 2005.

[14] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *KDD*, pages 557–566, 2009.

[15] S. Ma and J. Hellerstein. Mining partially periodic event patterns with unknown periods. In *ICDE*, pages 205 –214, 2001.

[16] R. Mijat. Better trace for better software. White paper, ARM, 2010. `http://www.arm.com/products/system-ip/debug-trace/index.php`.

[17] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *ICDE*, pages 412 –421, Feb 1998.

[18] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Inf. Syst.*, 24(1):25–46, Mar. 1999.

[19] C. Prada-Rojas, V. Marangozova-Martin, K. Georgiev, J.-F. Mehaut, and M. Santana. Towards a Component-Based Observation of MPSoC. In *ICPPW*, pages 542–549, 2009.

[20] STMicroelectronics. Orly SoC. `http://bit.ly/wUmu5Y`.

[21] STMicroelectronics. STi7200-MBoard platform. `http://bit.ly/z81nho`.

[22] R. Wille. The basic theorem of triadic concept analysis. *Order*, 12:149–158, 1995.

[23] J. Zou, J. Xiao, R. Hou, and Y. Wang. Frequent instruction sequential pattern mining in hardware sample data. In *ICDM*, pages 1205–1210, 2010.