Applying Sequential Pattern Mining to Portable RFID System Data

Heikki Sjöman, Martin Steinert
Department of Engineering Design and Materials
Norwegian University of Science and Technology
Trondheim, Norway
heikki.sjoman@ntnu.no, martin.steinert@ntnu.no

***Abstract*—This paper presents how data mining can be applied to RFID proximity tracking data captured in a production setting. The WINEPI algorithm is explained and used for mining sequential patterns from transaction data produced by portable RF transceivers that can be attached, for example, to the personnel and machines of a production facility. The contribution of this paper is the additional mindset of how data can be produced for a data warehouse with dedicated sensors in order to prototype the data warehouse itself – and how we can use this created knowledge as a help when designing intelligent manufacturing systems.**

Keywords—RFID;data mining; sequential patterns; intelligent manufaturing; industry 4.0; data warehouses

I. INTRODUCTION

The standard data warehouses are in a way statically related to the IT infrastructure as the data is often acquired through quite rigid operational processes [1]. Creating a new data warehouse is like developing a new product. It is context dependent and we cannot confirm what the final requirements of the system are and, according to Kimball & Ross, data warehouses should especially adapt to constant change. Furthermore, there is an irrational human as an end-user for the data warehouse business intelligence systems [1]. Reference [2] describes early stage product development as a multidisciplinary, iterative process that is ambiguous by nature [3][4]. There might be situations where one needs to create a new data warehouse quickly or to prototype data warehouses. Technically this might be challenging since data warehouses are traditionally based on data that is acquired from operational systems. For modern data warehouses in manufacturing and automation, there might be a need for a more dynamic approach of creating data sets. When we are designing systems where data warehouses are in the key roles, we would like to optimize the design thereof, but it is impossible since we do not have the data yet. For example, there are challenges and needs in the industry to deploy RFID systems, but it is not yet widely understood how the systems would work in all manufacturing contexts and what the benefits and costs, respectively, are [5]. We are proposing an approach in addition to traditional method of building data warehouses that is based on simulated or provided data. If we create our own data cube dynamically, we allow ourselves to explore different possibilities of creating the warehouse and learn while at the same time we can deploy a wide range of methods available in the data mining community. In this paper we give one example of how this approach can be applied, by building our own sensor-arrays in order to relatively quickly create a data warehouse, as well as gather test data. We build our own dedicated sensors by using existing technology, we deploy them, we generate data, we learn from it, and then we can use existing standard data algorithms. In the bigger picture we are getting in a position where we could be more independent of simulations of the suppliers, or at worst avoid the situation of designing and building systems without any data. This problem is often encountered in product development: it is hard to design for an imaginary, complex system and for its possible benefits while the development is kept on an abstract level [2]. Challenges like this could be overcome by initially pre-defining the data warehouse more dynamically and loosely and through prototyping learn about the future data structures of the new data warehouse before taking the investment decision.

We are on the way of developing a general comprehensive platform for gathering temporal relative proximity data. This platform is flexible and can be easily modified to accommodate new ideas and changes that appear in product development. In order to prove the usefulness of this system, we wanted to go through one full round of iteration in developing of a system from the beginning until the data is created and analyzed. We need tools to mine the produced data so that it can instantly show the value for its users. We define data mining as discovering interesting, and potentially useful patterns from data through applying algorithms in order to extract novel, previously hidden knowledge and relations [7]. The WINEPI algorithm is applied to the experimental dataset in order to test the possibilities of mining frequent sequential patterns [8]. Our main emphasis is on the approach of how data warehouses can be prototyped, tested, and how their requirements subsequently can be improved based on this early feedback for the overall system design, and therefore create extra value.

Following this introduction, the problem setting is introduced, the ways of solving it are discussed, and the data mining approach is described and applied. Finally, this paper concludes with results, discussions and future work.

II. Problem Setting

We have developed a portable system that consists of both active and passive RF transceivers. The active transceivers gather and store data, and the passive ones act as transponders respectively [9]. The devices are attached to different actors. For instance, members of a manufacturing team can carry the devices, or the devices can be fastened to objects, such as tools and materials. The dataset used here is from tracking a test group within an experiment. The raw data consists of proximity values and time stamps between different devices and is used as a proxy for human-human, human-machine and machine-machine interactions in order to model activities in sequences, for example, in workshop or factory settings. These sequences, to mention a few, might include frequent routes that are used in a factory, how often machines and resources are used, or in what spatial or temporal order interactions between these actors take place.

One of the devices (in prototype stage) is depicted in Fig. 1. It consists of basic electrical components and can be optimized greatly in respect to size and energy efficiency. The development thereof is not the focus, the devices rather show how to relatively fast and cheaply create a data warehouse, as well as actual data, and get feedback from it. The common next step is to apply data mining techniques to the data.
In this paper we chose the following goal: We want to detect frequent patterns and subsequently gain insights based on the data, from two or more actors within one event, that consists of four different distance ranges (0-1,5m; 1,5-3m; 3-5m; 5-11m) and a time stamp.

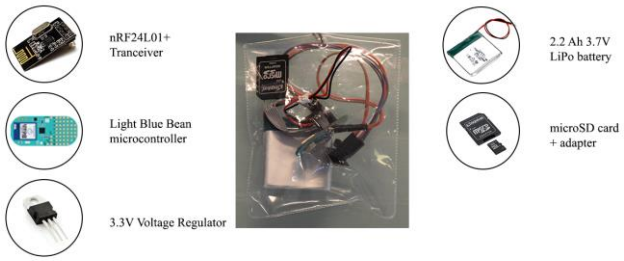

Fig. 1. *Parts of an active RF transceiver prototype [9].*

Our test data is from an experiment where a group of people uses facilities and workshops together or as individuals throughout one day. There were eight different sensors deployed as either static or active actors. The experiment day mainly consists of small meetings, as well as working in the workshop. Each sensor produce data point bursts every four seconds (+random time of 0-1000ms) from where the distance is estimated. In our experiment with eight device setup, depending on the activity level of a tagged person or equipment, each device produce approximately 28000 data points per day Plotting the data helps humans to understand the data gathered, but data mining reveals the possible underlying hidden patterns for the interpreter. We need a tool for recognizing sequential events and one is tested in the following chapters.

III. Computation Intelligence Approach

Because we are dealing with real-time RF systems, there is always some noise in the process of gathering data. This means that before starting the mining part, we need to pre-process the data and detect possible outliers. After this step, one can proceed to choose the algorithm. A graphical view of our raw data is given in Fig. 2. We wanted to deploy one of the original algorithms in sequential pattern mining to try out our approach. The WINEPI algorithm was originally developed for mining telecommunication network alarm logs in order to find sequential patterns from the data [8]. After a thorough research, this algorithm seemed to be a good alternative for mining interaction patterns from our data as well. Many of the algorithms created later are based on ideas of this sequential pattern-mining algorithm. With the WINEPI algorithm we can identify rules, compute their strengths and confidence intervals. It can detect both parallel and serial event episodes and can be applied with some modifications for multiple sequences [10].

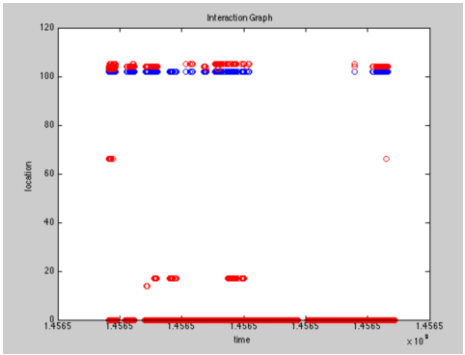

Fig. 2. Graphical view of our experimental data set of interaction data in a course of one day.

IV. Solution Procedure

In this section we define the formal concepts and notations that we will use to describe our mining methods. We also describe and explain the main points of the algorithm by using a descriptive pseudo algorithm. The solution is implemented in MATLAB R2014. Generally WINEPI algorithm moves a user-defined time window over the whole data set and count serial and parallel occurrences of actors present on a time stamp. After one pass over the data set it creates all the combinations of possible episodes for the next iteration, but prunes the candidates down based on the frequent patterns of

the preceding iteration. We follow the basic definitions introduced by Mannila, Toivonen and Verkamo in formally defining the original alarm sequences but modified to our context [3].

A. Definitions

*1) A **time point** is an integer that represents the occurrence time of the interaction between two devices.*

*2) **Time interval** is a continious sequence with a range of time points such that $[t_a, t_b] \equiv \{t: t_a \leq t \leq t_b\}$. The duration of the time interval can also be called as the **width** of the interval: $w = |t_b - t_a|$*

*3) Given a class of **actor types X**, an interaction point is a pair of terms (a,t) where $a \in X$ and t is the time point represented by a unix time stamp integer. An interaction **sequence** is an ordered set of interaction **events** defined as $S = \{(a_1,t_1),(a_2, t_2),...,(a_n,t_n)\}$, such that $a_i \in X$ for all $i=1,2,...,n$, and $t_i \leq t_{i+1}$ for all $i=1,...,n-1$.*

*4) Let C be a set of interaction **episodes** with respect to actor a_k. Let s be a set of the distinct interaction **events** a_1 $a_2,...,a_n$ such that $s \subseteq X$. When the threshold of minimum support is defined, the **episode** s is frequent if frequency (s) $\geq$ user defined minimum support.*

B. Main Algorithm

Episodes in the framework can be parallel, serial or combinations of these two. The first does not require a partial order in order to be recognized, but the second requires temporal order. Furthermore, each event must occur within a certain period of time. In practice, this is a user-defined time window that slides over the interaction sequence data and the algorithm is counting the occurrences of episodes within these windows. Frequency for each episode is calculated based on these counts of in how many windows the episode have been fully present out of the all windows over data. In other words, the algorithm finds all the sequential patterns that satisfy the predefined time constant, and whose frequency is exceeding a user set minimum frequency. The pseudo code for the main algorithm is presented in Fig. 3.

```
1. L1 = set of frequent events;
2. for ( k = 2; Lk-1 ≠ ∅; k++)
3.   Ck = New candidates of length k generated from Lk-1;
4.   Compute Lk , the set of frequent item-sets of length k;
5. end
```

Fig. 3. *Main algorithm of WINEPI [7]*

The algorithm runs multiple passes over the data. First it determines individual frequency count for each element in collection L_1 and counts the number of occurrences present in each window. It starts for the 1-element long episodes and continues to pass the data k times while always generating k-event-long candidate episodes C_k based on frequent patterns on the previous k-1 pass.

C. Generating and Pruning Candidate Episodes

In Fig. 4., the algorithm for creating the next iteration of collection of (parallel) candidate episodes is presented. The algorithm can be modified to produce serial or composite candidates as well [10]. Episodes are stored in a sorted manner that allows easier candidate generation. A group of episodes following each other of size k-1 that share the first k-2 events is called a block. In practice, the potential candidates are generated by first creating all combinations of two episodes inside of one block, and then pruning the non-frequent candidates. In order to make a more efficient identification of the blocks each first address of the episodes of a block is stored in the L_{k-1}.block_start array.

```
1. Ck = ∅;
2. l = 0;
   /* for one-event-long sequences the block size is one */
3. if k == 1
4.   for ( h = 1; h ≤ |Lk|; h++)
5.     Lk.block_start[h] = 1;
       /* go through all sequences in Lk-1 */
6. for ( i = 1; i ≤ |Lk-1|; i++)
7.   current_block_start = l+1;
     /* go through all distictive blocks in Lk-1 */
8.   for ( j = i; Lk-1.block_start[j] == Lk-1.block_start[i]; j++)
       /* Since Lk-1[i] and Lk-1[j] have k-2 first events in common,
          build a potential candidate s as their combination */
9.       for ( x = 1; x ≤ k-1 ; x++)
10.        s[x] = Lk-1[i][x];
11.      s[k] = Lk-1[j][k-1];
         /* For pruning build and test subsequences s' that do not
            contain s[x] */
12.      for ( y = 1; y < k-1 ; y++)
13.        for ( x = 1; x < y; x++)
14.          s'[x] = s[x];
15.        for ( x = y; x ≤ k-1; x++)
16.          s'[x] = s[x+1];
17.        if s' is not in Fk-1
18.          goto row 8; /* continue with the next j */
         /* All subsequences are in Fk-1, store s as candidate */
19.      l++;
20.      Ck[l] = s;
21.      Ck.block_start[l] = current_block_start;
22.  output Ck
```

Fig. 4. Algorithm for generating next candidate sequences *[10]*

D. Frequency Count of the Parallel Episodes

Counting the frequencies of the candidate episode is taking advantage of the fact that content of consecutive windows change only incrementally on each time step. The algorithm starts just before the interaction sequence and also ends just after the sequence, so that the user of the algorithm does not need to have any special treatment at the beginning or at the end of the sequence. Counting of sequential candidates works in principle so that every sequence has their own event.count attribute that increases every time an interaction point is added to the sliding window. When the event.count is the same length as the sequence itself it stores the starting time of a window to the attribute s.inwindow. When the event.count drops again, the frequency counter s.freq_count is increased by the number of windows the whole sequence was present in the window. In addition to normal storing of the candidates, the algorithm also stores references to the instances in the lists that are organized by interaction types and how many instances there are of each type. This means that interaction events added to the window will update the counter of all the references of that particular event. The pseudo code for counting frequencies is depicted in Fig. 5.

```
   /* Initialization */
1. for each s in C
2.   for each X in s
3.        X.count = 0;
4.        for ( i = 1; i ≤ |s|; i++ )
5.             contains(X,i) = ∅;
6. for each s in C
7.   for each X in s
8.        a = number of events of type X in s;
9.        contains(A,a) = contains(X,a)∪{s};
10.       s.event_count = 0;
11.       s.freq_count = 0;
   /* Regognition */
12.       for (start = T_s-win+1; T_s ≤ T_e; start++ )
   /* Bring in new events to the window */
13.            for all events (X,t) in s such that t=start+win-1
14.                 X.count++;
15.                 for each s∈contains(X,X.count)
16.                      s.event_count += X.count;
17.                      if s.event_count = |s|
18.                           s.inwindow = start;
           /* Drop out old events from the window */
19.            for all events (X,t) in s such that t = start-1
20.                 for each s∈contains(X,X.count)
21.                      if s.event_count = |s|
22.                           s.freq_count += start − s.inwindow;
23.                      s.event_count -= X.count;
24.                 X.count--;
   /* Output */
25.       for all sequences s in C
26.            if s.freq_count/(T_e-T_s+win-1) ≥ min_sup
27.                 output s;
```

Fig. 5. Algorithm for detecting parallel events *[10]*

E. Frequency Count of the Serial Episodes

Frequencies of serial episodes are counted by using automata that can exist in many instances at the same time. A new instance of automaton is initialized every time a first event of a sequence is arriving to the window. The automaton is removed when this same instance is exiting the window. The s.freq_count is increased again with the same principle of storing the s.inwindow every time when the whole sequence is completely in the window, and the automaton in its accepting state without other automata present in the same state, this will add to its frequency counter. The pseudo algorithm is described in Fig. 6.

For creating composite candidates of mixed serial and parallel events, a practical solution is to treat all the sequences like in parallel, but to check the correct partial ordering only when the whole sequence is present in the window.

V. Results and Discussions

From our data we are able to see how much time is spent among each devices, what kind of patterns there is and how frequently they occur. In this phase the nature of the data created was not that important since we just wanted test the system and learn. Far more important are the implications of how the system can be utilized when we are able to identify certain type of episodes from the data.

Another angle of assessing our problem could be to perceive it as a process discovery problem [11]. Similar algorithms are developed in this area of data mining and could be interesting to deploy them for our use as well.

```
/* Initialization */
1. for each s in C
2.    for ( i = 1; i ≤ |s|; i++ )
3.       s.initialized = 0;
4.       waits(s[i]) = ∅;
5. for each s in C
6.    waits(s[1]) = waits(s[1])∪{(s,1)};;
7.    s.freq_count = 0;
8. for ( t = T_s − win; t<T_s; t++)
9.    beginsat(t) = ∅;
   /* Regognition */
10.   for (start = T_s-win+1; start ≤ T_e; start++ )
   /* Bring in new events to the window */
11.      beginsat(start+win-1) = ∅;
12.      transitions = ∅;
13.      for all events (X,t) in s such that t = start+win-1
14.        for all (s,j)∈waits(X)
15.          if j=|s| and s.initialize[j] = 0
16.            s.inwindow = start;
17.          if j = 1
18.            transitions = transitions∪{(s,1,start+win-1)};
19.          else
20.            transitions = transitions∪{(s,j,s.initialized[j-
   1])};
21.            beginsat(s.initialized[j-1] /= {(s,1,j-1)};
22.            s.initialized[j-1] = 0;
23.            waits(X) /= {(s,j)};
24.      for all (s,j,t)∈transitions
25.        s.initialized[j] = t;
26.        beginsat(t) ∪= {(s,j)};
27.        if j < |s|
28.          waits(s[j+1]) ∪= {(s,j+1)};
       /* Drop out old events from the window */
29.      for all {(s,l)∈beginsat(start-1)
30.        if l = |s|
31.          s.freq_count += start-s.inwindow;
32.        else
33.          waits(s[l+1]) \= {(s,l+1)};
34.        s.initialized[l] = 0;
   /* Output */
35.   for all sequences s in C
36.      if s.freq_count / (T_e-T_s+win-1) ≥ min_sup
```

Fig. 6. Algorithm for detecting serial events [10]

VI. Conclusions

As a result of mining the testing day we got expected results. The mining works on the data like this and it is a good starting point for deploying other data analyzing tools for the future use of our system. With sequential pattern mining we are able to recognize different patterns, which could be, for example different actors using machines, actors spending time in a specified area, forklift lifting traceable pallets, identifying whether an actor is relatively attractive over certain threshold, defining social groups of a manufacturing facilities, optimizing production processes. Each round of prototyping with the possible data brings more knowledge what kind of data we want to produce and how we are going to apply it. We propose that through prototyping and using data mining we can create extra value as a low-cost step in the development process of data warehouses and intelligent manufacturing systems.

Acknowledgment

This research is supported by the Research Council of Norway (RCN) through its user-driven research (BIA) funding scheme, project number 236739/O30.

References

[1] R. Kimball, and M. Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[2] A. Gerstenberg, H. Sjöman, T. Reime, P. Abrahamsson, and M. Steinert. "A Simultaneous, Multidisciplinary Development and Design

Journey–Reflections on Prototyping." In *Entertainment Computing-ICEC 2015*, pp. 409-416. Springer International Publishing, 2015.

[3] W. Gaver, J. Beaver, and S. Benford. "Ambiguity as a resource for design." In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 233-240. ACM, 2003.

[4] L. Leifer, M. Steinert. Dancing with ambiguity: Causality behavior, design thinking, and triple-loop-learning. *Information Knowledge Systems Management*, *10*(1-4), 151-173. 2011.

[5] K. Wang, "Intelligent and integrated RFID (II-RFID) system for improving traceability in manufacturing." *Advances in Manufacturing* 2, no. 2: 106-120. 2014.

[6] M. Steinert, and L. J. Leifer. "'Finding One's Way': Re-Discovering a Hunter-Gatherer Model based on Wayfaring." *International Journal of Engineering Education* 28, no. 2: 251. 2012.

[7] A. Gosain, and A. Amar. "Security Issues in Data Warehouse: A Systematic Review." *Procedia Computer Science* 48: 149-157. 2015.

[8] H. Mannila, H. Toivonen, and A. I. Verkamo. "Discovery of frequent episodes in event sequences." *Data mining and knowledge discovery* 1, no. 3: 259-289.R. 1997.

[9] H. Sjöman, M. Steinert, G. Kress, and M. Vignoli. "Dynamically capturing engineering team interactions with wearable technology." In *DS 80-11 Proceedings of the 20th International Conference on Engineering Design (ICED 15) Vol 11: Human Behaviour in Design, Design Education; Milan, Italy, 27-30.07. 15*. 2015.

[10] J. Ahola. *Mining sequential patterns*. Vol. 10. VTT research report TTE1-2001, 2001.

[11] W. van der Aalst, A. Kalenkova, V. Rubin, and E. Verbeek. "Process discovery using localized events." In *Application and Theory of Petri Nets and Concurrency*, pp. 287-308. Springer International Publishing, 2015.