Technologie-Zentrum Informatik

Technical Report 37

Mining Temporal Patterns with WARMR

Andreas D. Lattner and Otthein Herzog

TZI-Bericht Nr. 37
2006

Universität Bremen

TZI-Berichte

Herausgeber:
Technologie-Zentrum Informatik
Universität Bremen
Am Fallturm 1
28359 Bremen
Telefon: +49-421-218-7272
Fax: +49-421-218-7820
E-Mail: info@tzi.de
http://www.tzi.de

ISSN 1613-3773

Mining Temporal Patterns with *WARMR*

Andreas D. Lattner and Otthein Herzog
TZI - Center for Computing Technologies, Universität Bremen
PO Box 330 440, D-28359 Bremen, Germany
e-mail: [adl|herzog]@tzi.de

September 27, 2006

Abstract

In the recent decades modern information technology has made possible to store and process huge amounts of data. Companies and organizations have identified the chances of collecting and utilizing data and the field of knowledge discovery in databases and data mining methods have attracted attention. In many domains temporal or sequential information plays an important role and thus, in the recent years many researchers also addressed temporal pattern mining. Many of the approaches take simple event or item sequences as learning input but some works also address events with temporal extent or sequences of relational data. In this work an approach to temporal pattern mining from time interval-based relational data is presented. Additionally, hierarchical class information can be provided for objects as well as for arguments in the relations. For support computation an observation time semantic as used by Höppner is used. We introduce a dynamic scene representation and show how it can be transferred in a way that it can be handled by the relational association rule mining algorithm *WARMR*. For a simple test scenario the created *WARMR* input files as well as the output are presented for illustration.

1 Introduction

Modern information technology allows for storing and processing huge amounts of data and thus the interest in taking advantage of the available data by applying data mining methods has increased in the last decades. While knowledge discovery in databases (KDD) has been defined as the whole "process of finding knowledge in data" including preprocessing and interpretation of the results, data mining is one step in this process where data mining methods are applied to actually search for patterns of interest [FPSS96]. In the case of temporal data mining temporal information is available in the data, e.g., annotated times when some items in transaction databases have been purchased or a sequential order of some events. In temporal data mining such temporal information shall be exploited in order to mine interesting rules including temporal information.

In the recent years, temporal data mining has been addressed by many researchers (cf. [ZSS03, LS06]). Many approaches to temporal pattern mining are based on Apriori like algorithms where patterns are created level-wise and only frequent patterns of one level are combined to form the next level candidates. The original Apriori algorithm mines association rules in transaction databases without time information [AS94]. It has been extended to mine sequential patterns from databases where the transactions have a timestamp. Many different approaches address the mining of patterns from such a representation or event sequences (without temporal extent of events), e.g., [AS95, SA96, MTV97]. In other works events can have a duration with start and end times and thus, temporal patterns with interval relations can be mined (e.g., [Höp03]). Another extension of association rule mining is the identification of association rules from more complex data than transactions. *WARMR*, for instance, mines association rules over multiple relations [DT99], and there are also works that can deal with both relational and sequential data (e.g., [Lee06]).

In many domains different kinds of objects can be in various relationships and events or actions can have a duration. Agents in dynamic environments, for instance, have to deal with complex situations including various temporal interrelations of actions and events. If more

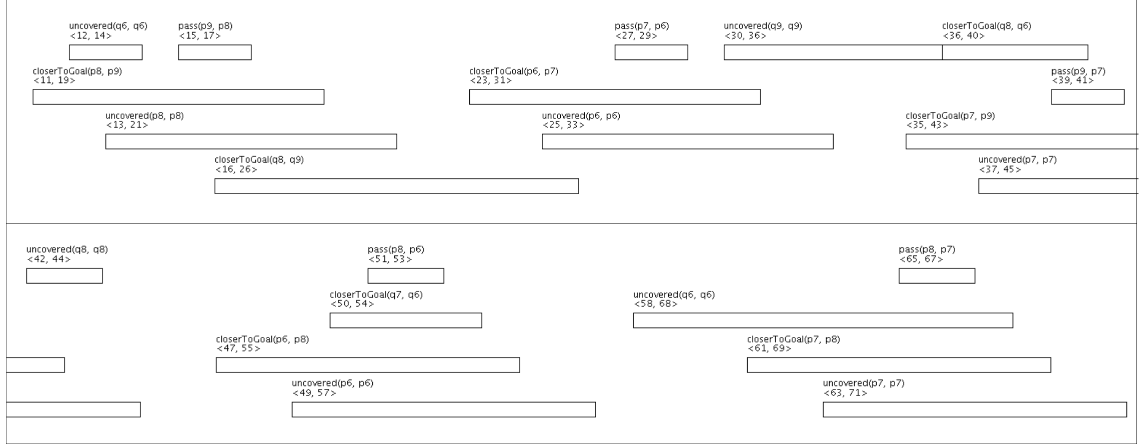

Figure 1: Test scenario

elaborated technologies like planning should be used, the representation of the agent's belief including background knowledge for its behavior decision can become very complex, too. It is necessary to represent knowledge about object classes and their properties, actual scenes with objects, their attributes and relations. If even more complex scenes with temporal extents shall be described this additional dimension must also be incorporated in the formalism. Fig. 1 shows an example sequence from the soccer domain. Here, time proceeds from left to right and the bars represent the duration of certain relations or actions. For instance, there is a player `p9` who passes the ball to player `p8` in the interval $\langle 15, 17 \rangle$. This example sequence is used throughout the paper for illustration purposes.

To the best of our knowledge there is no approach to temporal pattern mining learning from a relational and time interval-based representation and also taking class information into account in the mining process. In this report we present an interval-based representation for describing dynamic scenes and set up a temporal pattern mining task based on this representation. We show how *WARMR* [DT99] can be used to mine frequent temporal patterns from such a representation.

It should be mentioned that *WARMR* has already been used for sequential pattern mining tasks [DT99, JB01]. However, in these cases patterns have been mined from event sequences without temporal extension.

The next section provides some definitions and describes the problem to solve. In the subsequent section it is shown how *WARMR* can be used to mine the intended temporal patterns. After presenting a sample run with input and output data for *WARMR* the report ends with some concluding remarks.

2 Definitions and Problem Statement

The goal of the mining task is to find the set of all frequent temporal patterns from a dynamic scene as it is shown in Fig. 1. Before it is described how *WARMR* is used for the mining task we provide some definitions. Let $\mathcal{V}$, $\mathcal{O}$, $\mathcal{C}$, and $\mathcal{IR}$ be the sets of variables, objects, classes, and temporal interval relations, respectively.

Definition 2.1 (Dynamic Scene) *A dynamic scene is described by the 4-tuple* $ds = (\mathcal{P}, \mathcal{O}, i, \mathcal{DSS})$ *where* $\mathcal{P}$ *is the set of predicate instances,* $\mathcal{O}$ *is the set of objects in the dynamic scene,* $i : \mathcal{O} \rightarrow \mathcal{C}$ *maps objects to classes (instance-of relation), and* $\mathcal{DSS}$ *is the dynamic scene schema.* □

Definition 2.2 (Dynamic Scene Schema) *The schema of a dynamic scene* $\mathcal{DSS} = (\mathcal{C}, sc, \mathcal{PD}, \mathcal{IR})$ *consists of all schematic information.* $\mathcal{C}$ *is the set of classes and* $sc : \mathcal{C} \rightarrow \mathcal{C}$ *maps classes to their super classes and thus describes the class hierarchy.* $\mathcal{C}$ *consists of at least one element which denotes the most general class (object).* $\mathcal{PD}$ *is the set of predicate definitions and* $\mathcal{IR}$ *the set of the temporal interval relations.* □

Predicate definitions consist of the identifier, the arity, and the allowed ranges for the objects in their instances.

Definition 2.3 (Predicate Definition) *A predicate definition* pd *is defined as* $pd = (pd_{name}, pd_{arity}, pd_{classes})$ *with* $pd_{classes} = (c_1, c_2, \ldots, c_{pd_{arity}})$*. All* c_i *denote classes in the dynamic scene schema, i.e.,* $c_i \in \mathcal{C}$ *with* $1 \leq i \leq pd_{arity}$*.* □

Definition 2.4 (Predicate Instance) *Predicate instances* $pi = (pd, p_{objects}, \langle s, e \rangle)$ *are instances of predicate definition* pd*, consist of a list of object identifiers* $p_{objects} = (o_1, o_2, \ldots, o_{pd_{arity}})$ *with* $\forall o_i : o_i \in \mathcal{O}$ *of the dynamic scene, and additionally contain an interval of validity* $\langle s, e \rangle$ *with start time* s *and end time* e*.* □

For a better understanding we denote predicate instances in a more readable way: $holds(predicate(o_1, o_2, \ldots, o_{pd_{arity}}), \langle s, e \rangle)$ represents a predicate with $pd_{name} = predicate$, $p_{objects} = (o_1, o_2, \ldots, o_{pd_{arity}})$, start time s, and end time e. An example for a predicate in this notation is: $holds(inBallControl(p7), \langle 17, 42 \rangle)$.

Definition 2.5 (Interval Relation Function) *The interval relation function* $ir : \langle \mathbb{N}, \mathbb{N} \rangle \times \langle \mathbb{N}, \mathbb{N} \rangle \mapsto \mathcal{IR}$ *maps time interval pairs to interval relations.* □

It depends on the used interval relations $\mathcal{IR}$ how the actual mapping from the interval pairs to the interval relation has to be performed. Using, for instance, Allen's interval relations $ir(\langle s_1, e_1 \rangle, \langle s_2, e_2 \rangle) = b$ (*before*) if (and only if) $e_1 < s_2$ [All83].

An atomic pattern consists only of one predicate. The difference to predicate instances is that the list of arguments do not need to denote objects. In the general case the elements of the pattern are variables that can be bound to objects while pattern matching. However, it is also allowed to have arguments bound to objects in the pattern already.

Definition 2.6 (Atomic Pattern) *An atomic pattern is defined as* $p = (pd, p_{arg})$ *where* pd *denotes a predicate definition and* p_{arg} *specifies a list of terms* $p_{arg} = (v_1, v_2, \ldots, v_{pd_{arity}})$*. All* v_i *are either elements of* $\mathcal{O}$ *as defined in the dynamic scene or are elements of* $\mathcal{V}$*, the set of variables, i.e., it holds* $\forall v_i \in \mathcal{V} \cup \mathcal{O}$ *.* □

Definition 2.7 (Conjunctive Pattern) *A conjunction of atomic patterns is called conjunctive pattern. It connects the atomic patterns by a conjunction (logical AND):* $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ *where the* p_i *are atomic patterns with* $1 \leq i \leq n$*;* n *is called the size of the pattern.* □

Note that conjunctive patterns do have an implicit temporal order of their atomic patterns, i.e., each atomic pattern p_j must have the same or a greater start time than all its predecessor p_i with $i < j$. Similarly to the predicate instances above we introduce a short notation for conjunctive patterns: $predicate_1(v_{1_1}, \ldots, v_{1_{pd_{arity}}}) \wedge \ldots \wedge predicate_n(v_{n_1}, \ldots, v_{n_{pd_{arity}}})$. An example of a conjunctive pattern with two predicates is $uncovered(X) \wedge pass(Y, X)$.

Definition 2.8 (Class Restriction) *The class restriction defines for each variable* v_i *of a conjunctive pattern its least general class* c_i*. For a given variable list* $(v_1, v_2, \ldots, v_n)$ *the class restriction is represented by a class list* $(c_1, c_2, \ldots, c_n)$*.* □

Variable unifications define if certain variables in a (conjunctive) pattern should refer to the same object in the assignment during pattern matching, i.e., if variables are unified.

Definition 2.9 (Variable Unification) *A variable unification of a pattern* p *is defined as the unification of two different arguments* v_1 *and* v_2 *of one or two predicates of* p*, i.e., it must hold that* $v_1 = v_2$*.* □

B r_2 C A r_1 B	$<$	$<_c$	$\models$	$>_c$	$>$
$<$	$<$	$<$	$<$	$<, <_c, \models, >_c$	$<, <_c, \models, >_c, >$
$<_c$	$<, <_c$	$<, <_c$	$<_c$	$<_c, \models, >_c$	$<_c, \models, >_c, >$
$\models$	$<, <_c$	$<, <_c$	$\models$	$>_c$	$>$
$>_c$	$<, <_c$	$<, <_c, \models, >_c, >$	$>_c, >$	$>_c, >$	$>$
$>$	$<, <_c, \models, >_c, >$	$>_c, >$	$>_c, >$	$>_c, >$	$>$

Table 1: Composition table for the temporal relations

Binding a variable to a constant (i.e., to an instance) is denoted as instantiation:

Definition 2.10 (Instantiation) *A variable v_i is instantiated if it is bound to an instance of the set of objects in the dynamic scene, i.e., if $v_i = o$ with $o \in \mathcal{O}$.* □

A temporal restriction defines the constraints w.r.t. the validity intervals of two predicates in a conjunctive pattern. The order of the predicates in a pattern defines a temporal order implicitly already. A predicate must have an earlier or identical start time as all its succeeding predicates. Therefore, we define $\mathcal{IR}_{older} \subseteq \mathcal{IR}$ including those temporal relations where the start time of the first interval s_1 is before the start time of the second interval s_2, i.e., $s_1 < s_2$ and for the "head to head" temporal relations we define $\mathcal{IR}_{\models} \subseteq \mathcal{IR}$ where the start times are equal, i.e., $s_1 = s_2$.

Definition 2.11 (Temporal Restriction) *The temporal restriction $\mathcal{TR} = \{\mathcal{TR}[1,2], \ldots, \mathcal{TR}[n-1,n]\}$ of a conjunctive pattern p with size n is defined as the set of pairwise temporal relations between all atomic patterns. For each predicate pair $(pred_i, pred_j)$ of the pattern p where $pred_i$ appears before $pred_j$ in the pattern, i.e., $i < j$, the possible temporal relations between these two intervals are defined by the set $\mathcal{TR}[i,j]$. It must hold that $\forall tr_k \in \mathcal{TR}[i,j]$: $tr_k \in \mathcal{IR}_{older} \cup \mathcal{IR}_{\models}$ with $1 \leq i < n$ and $i < j \leq n$ due to the implicit temporal order of the predicates. If the name $pd_{name,j}$ of $pred_j$ is smaller than $pd_{name,i}$ of $pred_i$ w.r.t. a lexicographic order it must hold that $\forall tr_k \in \mathcal{TR}[i,j]$: $tr_k \in \mathcal{IR}_{older}$ in order to have a canonical representation of the sequences.* □

In the sample run described in section 4 we use just five temporal relations which can be seen as a condensed subset of the temporal relations introduced by [Fre92] and [All83]: before and after ($<, >$), older & contemporary and younger & contemporary ($<_c$, $>_c$), and head to head ($\models$). Thus, in our case $\mathcal{IR} = \{<, <_c, \models, >_c, >\}$, $\mathcal{IR}_{older} = \{<, <_c\}$, and $\mathcal{IR}_{\models} = \{\models\}$. The motivation for these temporal relations is due to keeping complexity low and still having the relevant temporal relations for setting up prediction rules. The composition table for these temporal relations is shown in Table 1.

Definition 2.12 (Temporal Pattern) *Temporal patterns $tp_i = (cp_i, \mathcal{TR}_i, cr_i)$ are defined as a 3-tuple of a conjunctive pattern $cp_i = ap_{i,1} \wedge ap_{i,2} \wedge \ldots \wedge ap_{i,size}$, a temporal restriction $\mathcal{TR}_i$, and a class restriction cr_i.* □

After having defined dynamic scenes, their schemata, and temporal patterns, we can define how to match such patterns to a dynamic scene. Pattern matching is essential for the computation of the support of a pattern. Basically, a match can be seen as a successful query to a database [Deh98]. In order to match a temporal pattern all predicates in the conjunction must be true (within a defined window size), the temporal restrictions between these predicates must be satisfied, and for the variable assignment the class restriction must not be violated.

Definition 2.13 (Pattern Match) *A match of pattern $p = (cp, tr, cr)$ is a valid assignment for each atomic pattern $p_i \in cp$ in the conjunctive pattern cp with size n to a corresponding (instantiated) predicate $p_{inst_i} \in \mathcal{P}$ of the dynamic scene where both predicate definitions of the atomic pattern $p_i = (pd_i, p_{i_{arg}})$ and the assigned predicate instance $p_{inst_i} = (pd_{inst_i}, p_{inst_{objects,i}}, \langle s_i, e_i \rangle)$ are identical, i.e., $pd_i = pd_{inst_i}$ and all arguments at the same*

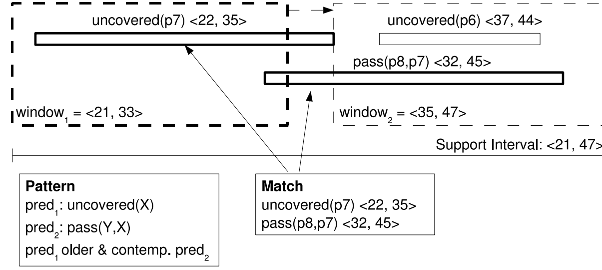

Figure 2: Pattern matching example

indices are pairwise unifiable. Furthermore, it must hold that no predicate instance is assigned more than once, i.e.: $\forall i, j : p_{inst_i} \neq p_{inst_j}$ *with* $i \neq j$ *and* $1 \leq i, j \leq n$.

Additionally, the match must be within the sliding window range. Let w_s *be the window's start position,* w *be the window size,* $w_e = w_s + w$ *be the window's end position, and* $\mathcal{P}_{match}$ *be the set of all predicate instances of the match. For all assigned predicate instances* $p_{inst_j} \in \mathcal{P}_{match}$ *with* $p_{inst_j} = (pd_{inst_j}, p_{inst_o bjects_j}, \langle s_j, e_j \rangle)$ *it must hold that* $s_j < w_e$ *and* $e_j \geq w_s$*, i.e., that the start time of the predicate instance has already passed and that it can still be seen within the window.*

Furthermore it must hold that none of the restrictions is violated. Let $\mathcal{O}_{match} = (o_1, o_2, \ldots, o_m)$ *be the list of objects in the assigned predicate instances and* $cr = (c_1, c_2, \ldots, c_m)$ *the class restriction of the pattern. Then it must hold that* $\forall i : instanceof_{trans}(o_i, c_i)$ *with* $1 \leq i \leq m$ *where* $instanceof_{trans}$ *is a transitive instance-of relation utilizing the class hierarchy defined by sc in* $\mathcal{DSS}$.

In order to satisfy the temporal restriction tr it must hold that $\forall r, s : ir(\langle s_r, e_r \rangle, \langle s_s, e_s \rangle) \in \mathcal{TR}[r, s]$ *with* $1 \leq r < n$ *and* $r < s \leq n$. □

As the frequency of a pattern is directly related to its support we first introduce how the support is computed in our case. In the task of frequent pattern discovery in logic, [Deh98] introduced an extra *key* parameter in order to determine what is counted. Entities are uniquely identified by each binding of the variables in key [Deh98, p. 34]. A disadvantage of this support definition is that the key parameter must be part of each pattern in order to get a support greater than zero. Thus, it is not possible to compare two different patterns if they do not share this key parameter.

We decided to use the observation time semantic for support computation as stated by Höppner. Here, the support is defined as "the total time in which (one or more) instances of P can be observed in the sliding window" [Höp03, p. 52]. The advantages of using observation time as support are the clear semantics and the better efficiency as not all matches have to be collected or maybe even further processed. The monotonicity property for this support definition holds and the support intervals of previous steps (i.e., of more general patterns) can be reused in order to restrict the search to parts of the temporal sequence in the subsequent levels.

Definition 2.14 (Support) *Let p be a temporal pattern, ds the dynamic scene, and* $\mathcal{M}$ *the set of matches. The validity interval of a single match* $m_i \in \mathcal{M}$ *is defined as* $v_i = [s_{max_i} - w + 1, e_{min_i} + w]$ *with* s_{max_i} *being the maximal start time and* e_{min_i} *the minimal end time of all predicate instances in* m_i*. The support of p w.r.t. ds is defined as the length of the union of all validity intervals of the matches:*
$supp(p) = length\left(\bigcup_{k=1}^{|\mathcal{M}|} v_k\right)$. □

This support definition computes the length of intervals where at least one match for a pattern can be found for a given window size. The frequency is the probability to find a match of a pattern at a random window position for a given dynamic scene and window size (cf. [Höp03]).

Algorithm 1 WARMR [DT99]

Input: Database $\mathbf{r}$; WRMODE language $\mathcal{L}$ and key; threshold $minfreq$
Output: All queries $Q \in \mathcal{L}$ with $frq(Q, \mathbf{r}, key) \geq minfreq$

1: Initialize level $d := 1$
2: Initialize the set of candidate queries $\mathcal{Q}_1 := \{? - key\}$
3: Initialize the set of infrequent queries $\mathcal{I} := \emptyset$
4: Initialize the set of frequent queries $\mathcal{F} := \emptyset$
5: **while** $\mathcal{Q}_d$ not empty **do**
6: Find $frq(Q, \mathbf{r}, key)$ of all $Q \in \mathcal{Q}_d$ using WARMR-EVAL
7: Move the queries $\in \mathcal{Q}_d$ with frequency below $minfreq$ to $\mathcal{I}$
8: Update $\mathcal{F} := \mathcal{F} \cup \mathcal{Q}_d$
9: Compute new candidates $\mathcal{Q}_{d+1}$ from $\mathcal{Q}_d$, $\mathcal{F}$, and $\mathcal{I}$ using WARMR-GEN
10: Increment d
11: **end while**
12: Return $\mathcal{F}$

Algorithm 2 WARMR-EVAL [DT99]

Input: Database $\mathbf{r}$; set of queries $\mathcal{Q}$; WRMODE key
Output: The frequencies of queries $\mathcal{Q}$

1: **for** each query $Q_j \in \mathcal{Q}$ **do**
2: Initialize frequency counter $q_j := 0$
3: **end for**
4: **for** each substitution $\theta_k \in answerset(? - key, \mathbf{r})$ **do**
5: Isolate the relevant fraction of the database $\mathbf{r}_k \subseteq \mathbf{r}$
6: **for** each query $Q_j \in \mathcal{Q}$ **do**
7: **if** query $Q_j\theta_k$ succeeds w.r.t. $\mathbf{r}_k$ **then**
8: Increment counter q_j
9: **end if**
10: **end for**
11: **end for**
12: **for** each query $Q_j \in \mathcal{Q}$ **do**
13: Return frequency counter q_j
14: **end for**

If the support value is divided by the sequence length of the dynamic scene plus the two times the window size minus one (sliding window at the start and the end of the sequence; the window must include the start time of the first interval in order to match a pattern) we get the frequency of the pattern, i.e., $freq(p) = \frac{supp(p)}{seq_{length}+2w-1}$.

Fig. 2 illustrates the matching of a pattern and the covered support interval by this match ($\langle 21, 47 \rangle$). The pattern in this examples matches the first time at window start position 21 when `pass(p8, p7) <32,45>` is visible in the window. It still matches as long as no end time point of a predicate in the match was left behind the window.

The goal of this work is to identify all frequent temporal patterns from a dynamic scene. The patterns language is defined as $\mathcal{L} = \{tp | tp = (cp, \mathcal{TR}, cr) \wedge freq(tp) \geq minfreq\} \cup \epsilon$ with $|cp| > 1$. The most general empty pattern is denoted by ϵ.

3 Mining Temporal Patterns with *WARMR*

As the temporal validity intervals of predicates can be seen as just another dimension of relations it should be possible to transfer the learning problem to relational association rule mining. Intuitively, it seems to be unhandy but feasible to add information about start and

Algorithm 3 WARMR-GEN [DT99]

Input: WRMODE language $\mathcal{L}$; infrequent queries $\mathcal{I}$; frequent queries $\mathcal{F}$; frequent queries $\mathcal{Q}_d$ for level d
Output: Candidate queries $\mathcal{Q}_{d+1}$ for level $d+1$

1: Initialize $\mathcal{Q}_{d+1} := \emptyset$
2: **for** each query $Q_j \in \mathcal{Q}_d$ and for each immediate specialization $Q'_j \in \mathcal{L}$ of Q_j **do**
3: Add Q'_j to $\mathcal{Q}_{d+1}$ unless:
4: (i) Q'_j is more specific than some query $\in \mathcal{I}$, or
5: (ii) Q'_j is equivalent to some query $\in \mathcal{Q}_{d+1} \cup \mathcal{F}$
6: **end for**
7: Return $\mathcal{Q}_{d+1}$

end time to every predicate and to regard it as additional dimensions. In this section we show how *WARMR* can be used to mine temporal patterns from the defined representation. The next subsection gives a brief introduction to *WARMR* and *ACE*. In the subsequent section a small example scene is set up. The following sections describe how this example can be transferred to *ACE* input in order to use *WARMR* for mining the patterns.

3.1 *WARMR* and *ACE*

As mentioned above *WARMR* extends Apriori for mining association rules over multiple relations [DT99, DT01]. The algorithms for the discovery of frequent Datalog patterns are shown in Listings 1-3 (adapted from [DT99, p. 15-18]). Agrawal et al. present algorithms for the generation of association rules in [AIS93, AS94]. Dehaspe and Toivonen [DT99, p. 19] introduce the notion of *query extensions* which are the "first order equivalent" of association rules. The created association rules are – similar to transaction-based association rules – expressions of the form $X \Rightarrow Y$ but here X and Y are sets of logical atoms of the form $p(t_1, \ldots, t_n)$ where each term t_i is a variable or a function (including constants as functions with arity 0). This more expressive representation allows for discovering rules like (cf. [DD97]):

$$likes(KID, A), has(KID, B) \Rightarrow prefers(KID, A, B)$$

Similar to the original association rule mining the task here is to discover all rules with a confidence about a minimum threshold *minconf* and a support above minimum support *minsup*. The difference is that the database is a deductive relational database instead of a simple transaction table.

The *WARMR* algorithm is based on Apriori but it exploits the lattice structure of atomsets instead of itemsets. The basic algorithm of *WARMR* is almost identical to Apriori. It was modified in the way examples are singled out and how the coverage test is performed [DD97]. The WARMR-gen algorithm extends the Apriori-gen algorithm by two pruning conditions where a theorem prover verifies if atomsets are contradictory or redundant.

ACE is a data mining system maintained by the Declarative Languages and Artificial Intelligence (DTAI) research group of the KU Leuven [BDD+02, BDR+06]. It provides a number of different relational data mining algorithms including *WARMR* [DT99]. We have used this system in our experiments.

3.2 Sequence and Schema Information

As shown in section 2 the description of dynamic scenes contains schematic information about the predicate definitions and their ranges as well as the dynamic predicates with the start and end times of the validity intervals. Fig. 3 represents the sample scene which is graphically represented in Fig. 1. The range definitions determine the classes for each argument of a predicate definition, i.e., instances of these predicates must have objects as arguments which are instances of the corresponding classes of the range definition. In the example the ranges of the

```
%Parameters
windowSize(12).
minFrequencyInPercent(10).
maxLevel(8).

%Predicate definitions
predicate(closerToGoal(_,_)).
predicate(pass(_,_)).
predicate(uncovered(_,_)).
range(closerToGoal, [object, object]).
range(pass, [object, object]).
range(uncovered,[object, object]).

%Class information
directSubClassOf(team1, object).
directSubClassOf(team2, object).
directInstanceOf(p6, team1).
directInstanceOf(p7, team1).
directInstanceOf(p8, team1).
directInstanceOf(p9, team1).
directInstanceOf(q6, team2).
directInstanceOf(q7, team2).
directInstanceOf(q8, team2).
directInstanceOf(q9, team2).
```

```
%Dynamic scene

holds(uncovered(q6, q6), 12, 14).
holds(pass(p9, p8), 15, 17).
holds(closerToGoal(p8, p9), 11, 19).
holds(uncovered(p8, p8), 13, 21).
holds(closerToGoal(q8, q9), 16, 26).
holds(pass(p7, p6), 27, 29).
holds(closerToGoal(p6,  p7), 23, 31).
holds(uncovered(p6, p6), 25, 33).
holds(uncovered(q9, q9), 30, 36).
holds(closerToGoal(q8, q6), 36, 40).
holds(pass(p9, p7), 39, 41).
holds(closerToGoal(p7, p9), 35, 43).
holds(uncovered(q8, q8), 42, 44).
holds(uncovered(p7, p7), 37, 45).
holds(pass(p8, p6), 51, 53).
holds(closerToGoal(q7, q6), 50, 54).
holds(closerToGoal(p6, p8), 47, 55).
holds(uncovered(p6, p6), 49, 57).
holds(pass(p8, p7), 65, 67).
holds(uncovered(q6, q6), 58, 68).
holds(closerToGoal(p7, p8), 61, 69).
holds(uncovered(p7, p7), 63, 71).
```

Figure 3: Sample input

three predicates `closerToGoal`, `pass`, and `uncovered` are set to the most general class `object`, Additionally, the description includes some parameters for learning, namely the window size, the support and the maximal refinement level for the mining process. The dynamic scene with the validity intervals are represented by `holds` relations with the predicate and the start and end time of the duration interval.

We developed a converter which automatically transfers this format to *ACE* input files. Different problems had to be solved in order to set up *WARMR* to mine the intended frequent patterns (with observation time support calculation). The following sections describe how the conversion is performed.

3.3 Transferring Class Information and Predicate Instances

The transformation of the class hierarchy and corresponding instances is straight forward. The `directSubClassOf` and `directInstanceOf` relations can be kept and put to *ACE*'s knowledge base file (Fig. 4). Transitive clauses for querying instances of classes and subclasses of a class can are defined in the background knowledge file (Fig. 7 and 8). The `holds` predicates representing the validity intervals of relations are now represented by relations with an additional argument which stands for the time interval. The predicate instance $holds(pass(p8, p7), \langle 32, 45\rangle)$ is converted to `pass(1, p8, p7, i(32, 45))` where the first argument is a unique ID for the predicate instances.

3.4 Refinement Operators

For setting up the learning bias in *WARMR* it is necessary to define *rmode* statements. These statements define how a query can be extended during the generation of new query candidates. Basically, a *rmode* consists of a predicate and specifications for the arguments, i.e., if an existing (`+`), new (`-`), or unique variable (`\`) or if a constant should be used. It is also possible to define constraints which must be satisfied in order to add an atom to the query. There are many other possibilities to specify the language bias which are out of scope of this report. More details can be found, for instance, in Dehaspe's doctoral thesis and the *ACE* user's manual [Deh98, BDR+06].

There are different refinements which have to be modelled by the rmodes. For lengthening (i.e., extending a query by a predicate) a *rmode* must be defined for each predicate definition.

```
directSubClassOf(team1, object).
directSubClassOf(team2, object).
directInstanceOf(p6, team1).
directInstanceOf(p7, team1).
directInstanceOf(p8, team1).
directInstanceOf(p9, team1).
directInstanceOf(q6, team2).
directInstanceOf(q7, team2).
directInstanceOf(q8, team2).
directInstanceOf(q9, team2).

uncovered(1, q6, q6, i(12, 14)).
pass(2, p9, p8, i(15, 17)).
closerToGoal(3, p8, p9, i(11, 19)).
uncovered(4, p8, p8, i(13, 21)).
closerToGoal(5, q8, q9, i(16, 26)).
pass(6, p7, p6, i(27, 29)).
closerToGoal(7, p6, p7, i(23, 31)).
uncovered(8, p6, p6, i(25, 33)).
uncovered(9, q9, q9, i(30, 36)).
closerToGoal(10, q8, q6, i(36, 40)).
pass(11, p9, p7, i(39, 41)).
closerToGoal(12, p7, p9, i(35, 43)).
uncovered(13, q8, q8, i(42, 44)).
uncovered(14, p7, p7, i(37, 45)).
pass(15, p8, p6, i(51, 53)).
closerToGoal(16, q7, q6, i(50, 54)).
closerToGoal(17, p6, p8, i(47, 55)).
uncovered(18, p6, p6, i(49, 57)).
pass(19, p8, p7, i(65, 67)).
uncovered(20, q6, q6, i(58, 68)).
closerToGoal(21, p7, p8, i(61, 69)).
uncovered(22, p7, p7, i(63, 71)).


currentIndex(0).
currentIndex(1).
currentIndex(2).
currentIndex(3).
currentIndex(4).
currentIndex(5).
currentIndex(6).
currentIndex(7).
currentIndex(8).
currentIndex(9).
currentIndex(10).
currentIndex(11).
currentIndex(12).
currentIndex(13).
currentIndex(14).
currentIndex(15).
currentIndex(16).
currentIndex(17).
currentIndex(18).
currentIndex(19).
currentIndex(20).
currentIndex(21).
currentIndex(22).
currentIndex(23).
currentIndex(24).
currentIndex(25).
currentIndex(26).
currentIndex(27).
currentIndex(28).
currentIndex(29).
currentIndex(30).
currentIndex(31).
currentIndex(32).
currentIndex(33).
currentIndex(34).
currentIndex(35).
currentIndex(36).
currentIndex(37).
currentIndex(38).
currentIndex(39).
currentIndex(40).
currentIndex(41).
currentIndex(42).
currentIndex(43).
currentIndex(44).
currentIndex(45).
currentIndex(46).
currentIndex(47).
currentIndex(48).
currentIndex(49).
currentIndex(50).
currentIndex(51).
currentIndex(52).
currentIndex(53).
currentIndex(54).
currentIndex(55).
currentIndex(56).
currentIndex(57).
currentIndex(58).
currentIndex(59).
currentIndex(60).
currentIndex(61).
currentIndex(62).
currentIndex(63).
currentIndex(64).
currentIndex(65).
currentIndex(66).
currentIndex(67).
currentIndex(68).
currentIndex(69).
currentIndex(70).
currentIndex(71).
currentIndex(72).
currentIndex(73).
currentIndex(74).
currentIndex(75).
currentIndex(76).
currentIndex(77).
currentIndex(78).
currentIndex(79).
currentIndex(80).
currentIndex(81).
currentIndex(82).
```

Figure 4: Generated *ACE* input file "sample.kb"

In order to avoid the same predicate instance being used more than once in a match it must be guaranteed that the predicate ID variable differs from all other predicate ID variables of this query. This can be forced by the backslash modifier. The *rmode* `rmode(closerToGoal(\Id, -X0, -X1, -I, +WindowStart))` says that a `closerToGoal` predicate can be added to the query where the `Id` must be unique (i.e., a new variable unequal to all existing ones), `X0`, `X1`, and `I` which represent the two players and the time interval must be new variables, and that `WindowStart` must be an existing variable. The `WindowStart` is used to check if a matched

```
typed_language(yes).
type(currentIndex(index)).
type(getWindowPos(index, starttime)).
type(before(interval, interval)).
type(olderContemp(interval, interval)).
type(headToHead(interval, interval)).

type(unif(object, object)).
rmode(unif(+X, +Y)).
constraint(unif(X,Y), not_occurs(unif(_,X))).
constraint(unif(X,Y), X\==Y).

type(closerToGoal(id, object, object, interval, starttime)).
rmode(closerToGoal(\Id, -X0, -X1, -I, +WindowStart)).
type(useful_constant_closerToGoal_O0(_)).
type(eq_obj_closerToGoal_O0(object, _)).
rmode(#(C: useful_constant_closerToGoal_O0(C), eq_obj_closerToGoal_O0(+X, C))).
constraint(eq_obj_closerToGoal_O0(X, Y), occurs(closerToGoal(_,X,_,_,_))).
constraint(unif(_,Y), not_occurs(eq_obj_closerToGoal_O0(Y, _))).
constraint(instanceOf(X,_), not_occurs(eq_obj_closerToGoal_O0(X, _))).
constraint(eq_obj_closerToGoal_O0(Y, _), not_occurs(unif(_,Y))).
constraint(eq_obj_closerToGoal_O0(Y, _), not_occurs(instanceOf(Y,_))).

type(useful_constant_closerToGoal_O1(_)).
type(eq_obj_closerToGoal_O1(object, _)).
rmode(#(C: useful_constant_closerToGoal_O1(C), eq_obj_closerToGoal_O1(+X, C))).
constraint(eq_obj_closerToGoal_O1(X, Y), occurs(closerToGoal(_,_,X,_,_))).
constraint(unif(_,Y), not_occurs(eq_obj_closerToGoal_O1(Y, _))).
constraint(instanceOf(X,_), not_occurs(eq_obj_closerToGoal_O1(X, _))).
constraint(eq_obj_closerToGoal_O1(Y, _), not_occurs(unif(_,Y))).
constraint(eq_obj_closerToGoal_O1(Y, _), not_occurs(instanceOf(Y,_))).

type(pass(id, object, object, interval, starttime)).
rmode(pass(\Id, -X0, -X1, -I, +WindowStart)).
type(useful_constant_pass_O0(_)).
type(eq_obj_pass_O0(object, _)).
rmode(#(C: useful_constant_pass_O0(C), eq_obj_pass_O0(+X, C))).
constraint(eq_obj_pass_O0(X, Y), occurs(pass(_,X,_,_,_))).
constraint(unif(_,Y), not_occurs(eq_obj_pass_O0(Y, _))).
constraint(instanceOf(X,_), not_occurs(eq_obj_pass_O0(X, _))).
constraint(eq_obj_pass_O0(Y, _), not_occurs(unif(_,Y))).
constraint(eq_obj_pass_O0(Y, _), not_occurs(instanceOf(Y,_))).

type(useful_constant_pass_O1(_)).
type(eq_obj_pass_O1(object, _)).
rmode(#(C: useful_constant_pass_O1(C), eq_obj_pass_O1(+X, C))).
constraint(eq_obj_pass_O1(X, Y), occurs(pass(_,_,X,_,_))).
constraint(unif(_,Y), not_occurs(eq_obj_pass_O1(Y, _))).
constraint(instanceOf(X,_), not_occurs(eq_obj_pass_O1(X, _))).
constraint(eq_obj_pass_O1(Y, _), not_occurs(unif(_,Y))).
constraint(eq_obj_pass_O1(Y, _), not_occurs(instanceOf(Y,_))).
```

Figure 5: Generated *ACE* input file "sample.s" (1/2)

predicate instance lies within the scope of the sliding window. More details about the support computation can be found in section 3.5.

Temporal relations between intervals are represented by clauses which check if the temporal relation actually holds for the interval pair, i.e., for each temporal relation (here: `before`, `olderContemp`, and `headToHead`) a clause exists and a *rmode* is created. In order to refine a pattern by adding a temporal constraint one of the temporal clauses is added to the query by relating two intervals of existing predicates of the query to each other. Thus, in the *rmode* specifications (e.g., `rmode(before(+I1, +I2))`) the two interval variables must be existing ones in order to set up a temporal constraint for them.

Unification is handled by a special unification clause (`unif(X, Y)`) which unifies two existing variables in the previous query. The *rmode* declarations of *ACE* also provide means to define *rmodes* which do not introduce a new variable in the new atom but reuse an existing one. However, our intended solution should also cover the instantiation of variables (i.e., using

```
type(uncovered(id, object, object, interval, starttime)).
rmode(uncovered(\Id, -X0, -X1, -I, +WindowStart)).
type(useful_constant_uncovered_O0(_)).
type(eq_obj_uncovered_O0(object, _)).
rmode(#(C: useful_constant_uncovered_O0(C), eq_obj_uncovered_O0(+X, C))).
constraint(eq_obj_uncovered_O0(X, Y), occurs(uncovered(_,X,_,_,_))).
constraint(unif(_,Y), not_occurs(eq_obj_uncovered_O0(Y, _))).
constraint(instanceOf(X,_), not_occurs(eq_obj_uncovered_O0(X, _))).
constraint(eq_obj_uncovered_O0(Y, _), not_occurs(unif(_,Y))).
constraint(eq_obj_uncovered_O0(Y, _), not_occurs(instanceOf(Y,_))).

type(useful_constant_uncovered_O1(_)).
type(eq_obj_uncovered_O1(object, _)).
rmode(#(C: useful_constant_uncovered_O1(C), eq_obj_uncovered_O1(+X, C))).
constraint(eq_obj_uncovered_O1(X, Y), occurs(uncovered(_,_,X,_,_))).
constraint(unif(_,Y), not_occurs(eq_obj_uncovered_O1(Y, _))).
constraint(instanceOf(X,_), not_occurs(eq_obj_uncovered_O1(X, _))).
constraint(eq_obj_uncovered_O1(Y, _), not_occurs(unif(_,Y))).
constraint(eq_obj_uncovered_O1(Y, _), not_occurs(instanceOf(Y,_))).

rmode_key(currentIndex(I)).
root(currentIndex(I)).
rmode(1:getWindowPos(+Index, WindowStart)).
rmode(before(+I1, +I2)).
rmode(olderContemp(+I1, +I2)).
rmode(headToHead(+I1, +I2)).
constraint(headToHead(X,Y), not(X=Y)).

type(class(_)).
type(instanceOf(object, _)).
rmode(#(C: class(C), instanceOf(+X, C))).

constraint(instanceOf(X,Y), not_occurs(instanceOf(X,_))).
constraint(instanceOf(X,Y), not_occurs(unif(_,X))).
constraint(unif(_,X), not_occurs(instanceOf(X,Y))).
minfreq(0.1).
warmr_maxdepth(8).
```

Figure 6: Generated *ACE* input file “sample.s” (2/2)

constants). Setting up *rmodes* for all cases (unification, constants, and new variables) and their combinations in predicates with an arbitrary (potentially large) number of arguments would have lead to a huge number of *rmodes* for the predicates. Thus, if a new predicate is added to the query all actual arguments of the relation are new variables in the beginning. These can be unified with another variable or can be bound to a constant in further refinement steps.

For instantiation a *rmode* definition allows a variable to be unified with an instance (e.g., `eq_obj_uncovered_O1(+X, C)` for the second argument of the `uncovered` predicate). The set of instance candidates depends on the predicate where the variable occurs. Only those instances are taken into account which appear at least in one of the predicates at the variable's position in the dynamic scene, i.e., no “impossible” query will be generated here. For each argument of each predicate definition a clause for finding the possible objects exist (e.g., `useful_constant_uncovered_O1(C)`). It is necessary to define all these different `useful_constant` clauses as the sets of relevant objects can differ for each argument.

Class refinement is performed by adding `instanceOf` predicates, constraining a variable to a certain class (or one of its sub classes). A constraint definition makes sure that for each variable just one `instanceOf` predicate will be added. Additional constraints ensure that a variable will be used just for instantiation or class refinement and that unified variables are not refined at all, e.g.,

```
constraint(unif(_,Y), not_occurs(eq_obj_uncovered_O1(Y, _))).
constraint(instanceOf(X,_), not_occurs(eq_obj_uncovered_O1(X, _))).
constraint(eq_obj_uncovered_O1(Y, _), not_occurs(unif(_,Y))).
constraint(eq_obj_uncovered_O1(Y, _), not_occurs(instanceOf(Y,_))).
```

```
unif(X, Y) :-
        X = Y.

createIndexList_(Start, End, List, List) :-
        End < Start, !.

createIndexList_(Start, End, ListPart, List) :-
        NewEnd is (End - 1),
        createIndexList_(Start, NewEnd, [End|ListPart], List).

createIndexList(Start, End, List) :-
        createIndexList_(Start, End, [], List).

getWindowPos(CurrentIndex, WindowPos) :-
        windowSize(WindowSize),
        FirstStart is (CurrentIndex - WindowSize),
        createIndexList(FirstStart, CurrentIndex, IndexList),
        member(WindowPos, IndexList).


validInterval(S, E, WindowStart) :-
        windowSize(WindowSize),
        WindowEnd is (WindowStart + WindowSize),
        E > WindowStart,
        S < WindowEnd.

before(i(_S1, E1), i(S2, _E2)) :-
        E1 < S2.

olderContemp(i(S1, E1), i(S2, _E2)) :-
        S1 < S2,
        E1 >= S2.

headToHead(i(S1, _E1), i(S2, _E2)) :-
        S1 == S2.

windowSize(12).

closerToGoal(Id, O0, O1,  i(S, E), WindowStart) :-
        closerToGoal(Id, O0, O1,  i(S, E)),
        validInterval(S, E, WindowStart).

useful_constant_closerToGoal_O0(C) :-
        findall(X, closerToGoal(_,X,_,_), ConstList), setof(Y, member(Y,ConstList), ConstSet), !,
        member(C, ConstSet).

eq_obj_closerToGoal_O0(X, Y) :-
        X = Y.

useful_constant_closerToGoal_O1(C) :-
        findall(X, closerToGoal(_,_,X,_), ConstList), setof(Y, member(Y,ConstList), ConstSet), !,
        member(C, ConstSet).

eq_obj_closerToGoal_O1(X, Y) :-
        X = Y.
```

Figure 7: Generated *ACE* input file “sample.bg” (1/2)

3.5 Support Computation

WARMR needs a counting attribute which is used for support computation, i.e., the number of different values of this attribute where a query matches determines the support of the query. In our case the support is defined to be the number of temporal positions where within a sliding window a pattern holds. In order to let *WARMR* compute the intended support a predicate `currentIndex` has been introduced and used as counting attribute. For each existing temporal position a predicate is created in the knowledge base file (Fig. 4). In combination with another predicate representing the window position (`getWindowPos`) for each temporal position it can be checked if a pattern holds. This is needed in order to check for each temporal position if

```
pass(Id, O0, O1,  i(S, E), WindowStart) :-
        pass(Id, O0, O1,  i(S, E)),
        validInterval(S, E, WindowStart).

useful_constant_pass_O0(C) :-
        findall(X, pass(_,X,_,_), ConstList), setof(Y, member(Y,ConstList), ConstSet), !,
        member(C, ConstSet).

eq_obj_pass_O0(X, Y) :-
        X = Y.

useful_constant_pass_O1(C) :-
        findall(X, pass(_,_,X,_), ConstList), setof(Y, member(Y,ConstList), ConstSet), !,
        member(C, ConstSet).

eq_obj_pass_O1(X, Y) :-
        X = Y.

uncovered(Id, O0, O1,  i(S, E), WindowStart) :-
        uncovered(Id, O0, O1,  i(S, E)),
        validInterval(S, E, WindowStart).

useful_constant_uncovered_O0(C) :-
        findall(X, uncovered(_,X,_,_), ConstList), setof(Y, member(Y,ConstList), ConstSet), !,
        member(C, ConstSet).

eq_obj_uncovered_O0(X, Y) :-
        X = Y.

useful_constant_uncovered_O1(C) :-
        findall(X, uncovered(_,_,X,_), ConstList), setof(Y, member(Y,ConstList), ConstSet), !,
        member(C, ConstSet).

eq_obj_uncovered_O1(X, Y) :-
        X = Y.

class(X) :-
        directSubClassOf(X,_).

subClassOf(X, Y) :-
        directSubClassOf(X, Y).

subClassOf(X, Y) :-
        directSubClassOf(Z,Y),
        subClassOf(X,Z).

instanceOf(Inst, Class) :-
        directInstanceOf(Inst, Class).

instanceOf(Inst, Class) :-
        directInstanceOf(Inst, SubClass),
        subClassOf(SubClass, Class).
```

Figure 8: Generated *ACE* input file "sample.bg" (2/2)

there is a sliding window position with a match that actually covers this position. If we only check for each time index if there is a match in the sliding window starting from this index we might miss matches of earlier sliding window positions that also cover the current index. Thus, we have to check for a match in all sliding window positions that cover the current index. The `validInterval` clause in the background knowledge (Fig. 7) checks for a potential predicate match if it is relevant w.r.t. the sliding window position.

4 Sample Run

Fig. 9 shows the *ACE* output after running *WARMR* with the generated input files *sample.s*, *sample.kb*, and *sample.bg* (shown in Fig. 4 - 8). As it can be seen the first two levels are needed

```
/** ------------------------------------------------
 ** Algorithm:   Warmr
 ** Output type: freq_queries.out
 ** Date:        9/19/2006 16:15:52
 ** ACE version: 1.2.8-b1
 ** Hardware:    adl4 : i686 running Linux
 ** ------------------------------------------------
 **/

sample_size(83.0).
min_rel_freq(0.1).
min_abs_freq(8.3).

level(1).
freq(1,1,[currentIndex(A)],1.0).
c_counter(1,1).
f_counter(1,1).

level(2).
freq(2,1,[currentIndex(A),getWindowPos(A,B)],1.0).
c_counter(2,1).
f_counter(2,1).

level(3).
freq(3,1,[currentIndex(A),getWindowPos(A,B),closerToGoal(C,D,E,F,B)],0.975903614457831).
freq(3,2,[currentIndex(A),getWindowPos(A,B),pass(C,D,E,F,B)],0.903614457831325).
freq(3,3,[currentIndex(A),getWindowPos(A,B),uncovered(C,D,E,F,B)],0.987951807228916).
c_counter(3,3).
f_counter(3,3).

level(4).
freq(4,1,[currentIndex(A),getWindowPos(A,B),closerToGoal(C,D,E,F,B),closerToGoal(G,H,I,J,B),not(G=C)],
     0.746987951807229).
freq(4,2,[currentIndex(A),getWindowPos(A,B),closerToGoal(C,D,E,F,B),eq_obj_closerToGoal_OO(D,p6)],
     0.662650602409639).

(...)

level(8).
freq(8,1,[currentIndex(A),getWindowPos(A,B),closerToGoal(C,D,E,F,B),closerToGoal(G,H,I,J,B),not(G=C),unif(I,E),
     closerToGoal(K,L,M,N,B),not(K=C),not(K=G),closerToGoal(O,P,Q,R,B),not(O=C),not(O=G),not(O=K),
     eq_obj_closerToGoal_OO(P,p6)],0.156626506024096).

(...)

freq(8,47019,[currentIndex(A),getWindowPos(A,B),closerToGoal(C,D,E,F,B),pass(G,H,I,J,B),not(G=C),unif(I,D),
     unif(E,H),uncovered(K,L,M,N,B),not(K=C),not(K=G),olderContemp(N,J)],0.903614457831325).

(...)

freq(8,96613,[currentIndex(A),getWindowPos(A,B),uncovered(C,D,E,F,B),uncovered(G,H,I,J,B),not(G=C),
     instanceOf(I,team1),instanceOf(H,team1),instanceOf(E,team2),instanceOf(D,team2)],0.939759036144578).
freq(8,96614,[currentIndex(A),getWindowPos(A,B),uncovered(C,D,E,F,B),uncovered(G,H,I,J,B),not(G=C),
     instanceOf(I,team2),instanceOf(H,team2),instanceOf(E,team2),instanceOf(D,team2)],0.204819277108434).
c_counter(8,105235).
f_counter(8,96614).
```

Figure 9: Part of the *ACE* output file "sample.freq_queries.out"

for setting up the structure for the support computation (`currentIndex(A),getWindowPos(A,B)`). In the third level the initial patterns consisting of just one predicate are generated (e.g., `currentIndex(A),getWindowPos(A,B),closerToGoal(C,D,E,F,B)`). These patterns are refined in the further steps by the refinement operators defined above. For instance, the pattern

```
freq(8,47019,[currentIndex(A),getWindowPos(A,B),closerToGoal(C,D,E,F,B),
     pass(G,H,I,J,B),not(G=C),unif(I,D),unif(E,H),uncovered(K,L,M,N,B),
     not(K=C),not(K=G),olderContemp(N,J)],0.903614457831325).
```

says that there must be three predicates, namely `closerToGoal(D,E)`, `pass(H,I)` and `uncovered(L,M)` where `D` and `I` are unified as well as `E` and `H`, and where the temporal constraint that

`uncovered(L,M)` must be older and contemporary in relation to `pass(H,I)` is satisfied. The support of this pattern is 0.904.

The result file shows (we just present a snippet in Fig. 9) that the number of created patterns grows quite fast. In the eighth refinement level more than 100000 patterns have been created. The number of created patterns depends on the minsupport threshold and additionally, in the small example the support for many patterns is quite high.

5 Conclusion

In this report we gave a motivation for more complex descriptions of dynamic scenes and temporal patterns including event duration and hierarchical class information to be mined from such scenes. We showed that the validity intervals of predicates can be represented by two additional arguments in relations, and how the relational association rule mining algorithm *WARMR* – to the best of our knowledge – can be used for mining such temporal patterns.

Although mining the intended temporal patterns is possible, there are some drawbacks of the created solution. First of all, redundant patterns with identical matches and patterns which cannot have a support greater than zero are created. This could be avoided by exploiting the implicit information about variables' classes and temporal interrelations, i.e., by using a composition table as shown in Table 1 to remove all impossible temporal relations due to already known temporal relations between predicates. Avoiding redundancy and avoiding to check or to create "impossible" pattern candidates at all would be a great help to reduce complexity.

Another drawback is the current matching process. The realization of the sliding window by an extra predicate does not take advantage of the sequential structure of the dynamic scene representation. If just the currently visible predicates are taken into account at pattern matching the support computation should be faster as no irrelevant predicate instances have to be checked. However, it should be stated that *WARMR* is a generic approach to relational association rule mining and not specialized to temporal representations. The temporal data mining approach *MiTemP* [LH06] exploits information about temporal relations as well as hierarchical class information and addresses the problems identified here.

Acknowledgment

We would like to thank Frank Höppner at the Fachhochschule Braunschweig/Wolfenbüttel, Germany, for helpful discussions on support computations in temporal pattern mining. We also want to express our gratitude to the members of the Declarative Languages and Artificial Intelligence (DTAI) research group of the Katholieke Universiteit Leuven, Belgium, for providing the *ACE* system including *WARMR* [BDD+02]. Particularly, we would like to thank Jan Struyf for his great support with *ACE*/*WARMR*.

References

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499, September 1994.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, March 1995.

[BDD+02] Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Rammon, and Henk Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

[BDR+06] Hendrik Blockeel, Luc Dehaspe, Jan Rammon, Jan Struyf, Anneleen Van Assche, Celine Vens, and Daan Fierens. *The ACE Data Mining System, User's Manual.* Katholieke Universiteit Leuven, Belgium, February 16 2006.

[DD97] Luc Dehaspe and Luc De Raedt. Mining association rules in multiple relations. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, pages 125–132. Springer-Verlag, 1997.

[Deh98] Luc Dehaspe. *Frequent Pattern Discovery in First-Order Logic.* PhD thesis, Katholieke Universiteit Leuven, Belgium, 1998.

[DT99] Luc Dehaspe and Hannu Toivonen. Discovery of frequent DATALOG patterns. *Data Mining and Knowledge Discovery*, 3(1):7 – 36, March 1999.

[DT01] Luc Dehaspe and Hannu Toivonen. Discovery of relational association rules. In Sasǒ Džeroski and Nada Lavrač, editors, *Relational Data Mining*, pages 189 – 208. Springer-Verlag New York, Inc., 2001.

[FPSS96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery: An overview. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 1–34. MIT Press, 1996.

[Fre92] Christian Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54(1–2):199–227, 1992.

[Höp03] Frank Höppner. *Knowledge Discovery from Sequential Data*. PhD thesis, Technische Universität Braunschweig, 2003.

[JB01] Nico Jacobs and Hendrik Blockeel. From shell logs to shell scripts. In C. Rouveirol and M. Sebag, editors, *ILP 2001*, volume 2157 of *LNAI*, pages 80–90. Springer-Verlag Berlin Heidelberg, 2001.

[Lee06] Sau Dan Lee. *Constrained Mining of Patterns in Large Databases.* PhD thesis, Albert-Ludwigs-Universität Freiburg, 2006.

[LH06] Andreas D. Lattner and Otthein Herzog. Constraining the search space in temporal pattern mining. To be presented at the KDML 2006 Workshop of the special interest group of the GI on Knowledge Discovery, Data Mining and Machine Learning, Hildesheim, October 11-13 2006.

[LS06] Srivatsan Laxman and P. S. Sastry. A survey of temporal data mining. *SADHANA - Academy Proceedings in Engineering Sciences*, 31(2):173–198, April 2006.

[MTV97] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.

[SA96] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th International Conference on Extended Database Technology*, March 1996.

[ZSS03] Qiankun Zhao and Bhowmick Sourav S. Sequential pattern mining: A survey. Technical Report 2003118, CAIS, Nanyang Technological University, Singapore, 2003.