Constraining the Search Space in Temporal Pattern Mining

Andreas D. Lattner and Otthein Herzog
TZI – Center for Computing Technologies, Universität Bremen
PO Box 330 440, D-28334 Bremen, Germany
{adl|herzog}@tzi.de

Abstract

Agents in dynamic environments have to deal with complex situations including various temporal interrelations of actions and events. Discovering frequent patterns in such scenes can be useful in order to create prediction rules which can be used to predict future activities or situations. We present the algorithm *MiTemP* which learns frequent patterns based on a time interval-based relational representation. Additionally the problem has also been transfered to a pure relational association rule mining task which can be handled by *WARMR*. The two approaches are compared in a number of experiments. The experiments show the advantage of avoiding the creation of impossible or redundant patterns with *MiTemP*. While less patterns have to be explored on average with *MiTemP* more frequent patterns are found at an earlier refinement level.

1 Introduction

Agents in dynamic environments have to deal with complex situations including various temporal interrelations of actions and events. If more 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.

Discovering frequent patterns in dynamic scenes can be useful in order to create prediction rules which can be used to predict future activities of other agents or to predict future situations and, thus, adapt the own behavior by taking into account this additional knowledge. In previous work a relational representation with temporal validity intervals and algorithms for mining temporal patterns have been introduced in [Lattner *et al.*, 2006]. Here, we present a new algorithm which is also based on such a representation but avoids the creation of redundant patterns by defining an optimal refinement operator similar to the one in [Lee, 2006]. Additionally to the own implementation the problem has also been transfered to a pure relational association rule mining task which can be handled by *WARMR* [Dehaspe and Toivonen, 1999].

2 Related Work

Association rule mining addresses the problem of discovering association rules in data. One typical example is the mining of rules in basket data [Agrawal *et al.*, 1993]. Different algorithms have been developed for the mining of association rules in item sets (e.g., Apriori [Agrawal and Srikant, 1994]). [Mannila *et al.*, 1997] extended association rule mining by taking event sequences into account. They describe algorithms which find all relevant episodes which occur frequently in the sequence. Höppner presents an approach for learning rules about temporal relationships between labeled time intervals [Höppner, 2003]. The time intervals consist of propositional events and temporal relations are described by Allen's interval logic [Allen, 1983].

Dehaspe and Toivonen combine association rule mining algorithms with ILP techniques. Their system *WARMR* is an extension of Apriori for mining association rules over multiple relations [Dehaspe and Toivonen, 1999]. The generated rules consist of sets of logical atoms. This more expressive representations (compared to itemset mining) allows for discovering relational association rules.

[Kaminka *et al.*, 2003] introduce an approach which creates a sequence of certain events or behaviors from objects' positions in RoboCup soccer matches and searches for frequent sequences in the data. In their work they compare two approaches based on frequency counts and statistical dependencies. The events in the sequences do not have a temporal extent and the learned patterns do not abstract from concrete objects in the events.

[Lee, 2006] presents an approach to mine first-order logical (*SeqLog*) patterns from sequential relational data. He defines optimal refinement operators and algorithms for finding all frequent patterns. The support is defined by the number of event sequences that match the pattern.

The approaches of Höppner, Kaminka et al., and Lee are quite similar to the one presented here. In contrast to Höppner and Kaminka our approach can learn relational patterns with variables which is also supported by [Lee, 2006]. Like Höppner we take an interval-based representation as input and mine temporal patterns with temporal inter-relations between these intervals. We also use a similar support definition which is based on the probability to find a pattern at a random sliding window position in the sequence. In contrast to our work, Kaminka et al. and Lee's approaches are based on event sequences without temporal extent.

Our approach combines and extends these existing approaches. To the best of our knowledge no approach has addressed the mining of frequent temporal patterns from multi-relational time interval-based data. Our approach allows for taking hierarchical class information into account

(while existing approaches just provide types for variables). Reasoning techniques are used to exploit the knowledge about temporal relations and about classes in order to reduce the number of patterns to be generated and to avoid checking inconsistent patterns.

3 Definitions and Problem Statement

The goal of the mining task is to find the set of all frequent temporal patterns from a dynamic scene. Before the approach is described in detail 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 3.1 (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 the objects to classes (instance-of relation), and $\mathcal{DSS}$ is the dynamic scene schema.* □

Definition 3.2 (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 3.3 (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 3.4 (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 3.5 (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) $e_1 < s_2$ [Allen, 1983].

An atomic pattern consists only of one predicate pattern. 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 3.6 (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 3.7 (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.* □

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 3.8 (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 3.9 (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 $v_1 = v_2$.* □

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

Definition 3.10 (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 3.11 (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 predicates. 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 experiments described in section 6 we use just five temporal relations which can be seen as a condensed subset

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

of the temporal relations introduced by [Freksa, 1992] and [Allen, 1983]: 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 3.12 (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 [Dehaspe, 1998]. 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 3.13 (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 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 $\mathcal{DSS}$.

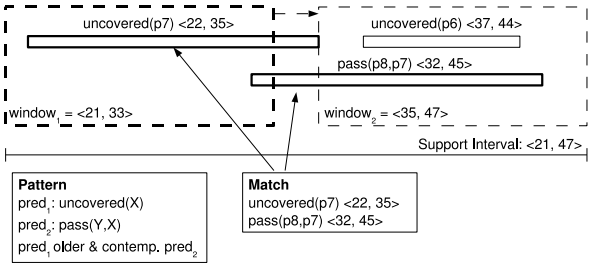

Figure 1: Pattern matching example

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, [Dehaspe, 1998] introduced an extra *key* parameter in order to determine what is counted. Entities are uniquely identified by each binding of the variables in key [Dehaspe, 1998, 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 > 0. 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" [Höppner, 2003, 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 3.14 (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. [Höppner, 2003]).

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. 1 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.

Algorithm 1 *MiTemP-main* (Pattern Generation)

Input: $ds = (\mathcal{P}, \mathcal{O}, i, \mathcal{DSS})$, win_{size}, $size_{min}$, $size_{max}$, $minfreq$ /* dynamic scene, window size, minimal and maximal pattern size, minimal frequency */
Output: All frequent patterns $\mathcal{P}_{freq}$ with $size$, $size_{min} \leq size \leq size_{max}$

1: Init $\mathcal{P}_{freq} = \emptyset, i = 1$
2: $C_i \leftarrow create\_single\_predicate\_patterns()$ /* Create one candidate for each predicate definition */
3: **while** $C_i \neq \emptyset$ **do**
4: $support[|C_i|] \leftarrow$ MiTemP-support(ds, win_{size}, C_i)
5: $L_i = \{c_j \in C_i | \frac{support(c_j)}{max\_supp} \geq minfreq\}$
6: $\mathcal{P}_{freq} \leftarrow \mathcal{P}_{freq} \cup \{l \in L_i | size_{min} \leq size(l) \leq size_{max} \wedge complete\_temporal\_restriction(l)\}$
7: $i \leftarrow i + 1$
8: $C_i =$ *MiTemP-gen-lengthening*(L_{k-1}) $\cup$ *MiTemP-gen-temp-refinement*(L_{k-1}) $\cup$ *MiTemP-gen-unification*(L_{k-1}) $\cup$ *MiTemP-gen-class-refinement*(L_{k-1}) $\cup$ *MiTemP-gen-instantiation*(L_{k-1})
9: **end while**

Algorithm 2 *MiTemP-gen-lengthening* (Lengthening Candidate Generation)

Input: $\mathcal{L}_{i-1}$ /* Frequent patterns of the previous step */
Output: New candidate patterns C_i

1: $\mathcal{F}_{i-1} = \{l \in \mathcal{L}_{i-1} | lengthening\_allowed(l)\}$
2: **for** $(p_i \in \mathcal{F}_{i-1})$ **do**
3: **for** $(p_j \in \mathcal{F}i - 1), j \geq i$ **do**
4: **if** $p_i = (ap_{i,1} \wedge ap_{i,2} \wedge \ldots \wedge ap_{i,i-2} \wedge ap_{i,i-1}) \wedge p_j = (ap_{i,1} \wedge ap_{i,2} \wedge \ldots \wedge ap_{i,i-2} \wedge ap_{j,i-1})$ **then**
5: $p_{new1} = (ap_{i,1} \wedge \ldots \wedge ap_{i,i-2} \wedge ap_{i,i-1} \wedge ap_{j,i-1})$
6: $p_{new2} = (ap_{i,1} \wedge \ldots \wedge ap_{i,i-2} \wedge ap_{j,i-1} \wedge ap_{i,i-1})$
7: /* Add if all subsets are frequent (prune step) */
8: **if** $\forall p_{sub} \subset p_{new1} : p_{sub} \in \mathcal{L}_{i-1}$ **then**
9: $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup p_{new1}$
10: **end if**
11: **if** $\forall p_{sub} \subset p_{new2} : p_{sub} \in \mathcal{L}_{i-1}$ **then**
12: $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup p_{new2}$
13: **end if**
14: **end if**
15: **end for**
16: **end for**

The goal of this work is to identify all frequent temporal patterns from a dynamic scene. In order to restrict the search space we introduce upper and lower limits for the number of predicates, i.e., for the minimal and maximal size of the conjunctive pattern, and force the temporal restriction to be completely constrained, i.e., all temporal relation sets must consist of exactly one element. If a pattern is frequent and it satisfies these conditions we refer to it as a *relevant frequent pattern*. The set of these patterns forms the language $\mathcal{L}_{MiTemP} = \{tp | tp = (cp, \mathcal{TR}, cr) \wedge freq(tp) \geq minfreq \wedge size_{min} \leq |cp| \leq size_{max} \wedge \forall i, j : |\mathcal{TR}[i, j]| = 1$ with $1 \leq i < |cp|$ and $i < j \leq |cp|\} \cup \epsilon$ with $|cp| > 1$. The most general empty pattern is denoted by ϵ.

4 *MiTemP*: Mining Temporal Patterns

This section introduces the *MiTemP* (Mining Temporal Patterns) algorithms. All algorithms are shown in pseudo code while the implementation has been realized with *XSB Prolog* [Sagonas *et al.*, 2006]. The main loop for the level-wise refinement is shown in Algorithm 1 (*MiTemP-main*). As mentioned above temporal patterns consist of different components like the conjunctive pattern, temporal restrictions, variable unification, class restrictions, and instantiations. For each of these components refinement operators exist that specialize a given pattern. In order to set up an optimal refinement operator which creates every pattern only once a status about the executed refinements is affixed to each pattern as it is also done by [Lee, 2006]. The status keeps track of how many refinements of each type have been performed and which position (predicate pair for temporal refinement, variable position for unification, class restriction, or instantiation) has been processed at last. A status $status(p) = (l, t, t_{last}, u, u_{last}, c, c_{last}, i, i_{last})$ where l, t, u, c, i are the number of refinement operations of the different types, namely lengthening, temporal refinement, unification, class refinement, and instantiation; $t_{last}, u_{last}, c_{last}, i_{last}$ refer to the last position where a temporal refinement, unification, class refinement, or instantiation has been performed. Similar to [Lee, 2006] we define the following refinement operations:

- Lengthening $\rho_L(p)$: Adding an atom to the end of a conjunctive pattern
- Temporal refinement $\rho_T(p)$: Adding a temporal constraint between two predicates in the conjunctive pattern at the leftmost position after the previous temporal refinement
- Unification $\rho_U(p)$: Unifying a variable v_j with a previous one v_i $(i < j)$ in the conjunctive pattern where no variable v_k with $k > j$ has been unified before
- Class refinement $\rho_C(p)$: Specializing a class c_i in the class restriction for the variables of the conjunctive pattern where no class restriction has been performed to any c_j with $j > i$.
- Instantiation $\rho_I(p)$: Instantiating a variable v_i of the conjunctive pattern with an instance $o \in \mathcal{O}$ where no variable v_j has been instantiated with $j > i$. It must also hold, that no variable v_k with $k \neq i$ has been instantiated to o.

The refinement operator is defined as the union of these operators: $\rho(p) = \rho_L(p) \cup \rho_T(p) \cup \rho_U(p) \cup \rho_C(p) \cup \rho_I(p)$. While [Lee, 2006] also defines a "deepening" operator (for replacing a variable by a functor) which is omitted here we introduce the class refinement operator which exploits the class hierarchy of the dynamic scene schema. Another difference is that the temporal refinement here adds arbitrary temporal relatios between time intervals while the "promotion" operator of Lee replaces the *before* relation between two events by a *directly before* relation.

Certain rules for each refinement coordinate when which refinement step is allowed. The lengthening operation is just allowed as long as no other refinement type has been applied and the maximal size of the conjunctive pattern is not exceeded. In order to perform a temporal refinement the minimum pattern size must be met, and no other refinement (except lengthening) must have been applied to the pattern. As we are looking for temporally completely constrained patterns the temporal refinement is only allowed to refine the next not yet processed predicate pair in the sequence. In the refinement step itself one of the possible temporal relations $tr \in \mathcal{TR}[i, j]$ is selected. After this step the composition table (Table 1) is used to further restrict the following temporal relations. Only those patterns where all predicate pairs are restricted to one temporal relation are further processed by other refinement types.

Algorithm 3 *MiTemP-support* (Support Computation)

Input: $ds = (\mathcal{P}, \mathcal{O}, i, \mathcal{DSS})$, win_{size}, $\mathcal{PL}$ /* dynamic scene, window size, pattern list */
Output: Support values $support(p_i)$ for all patterns $p_i \in \mathcal{PL}$

/* s_{min} is the earliest start and e_{max} is the latest end time */
Init $supp\_intervals(p_i) = \emptyset$, $next\_to\_check(p_i) = -\infty$
Init $\mathcal{P}_{win} = \emptyset$, $w_{start} = s_{min} - win_{size} + 1$
while $w_{start} \leq e_{max}$ **do**
  for $p_i \in \mathcal{PL}$ **do**
    if $(potential\_match(p_i, w_{start}) \wedge next\_to\_check(p_i) \leq w_{start})$ **then**
      $m \leftarrow pattern\_match(p_i, w_{start}, win_{size})$
      if $m \neq null$ **then**
        $supp\_intervals(p_i) \leftarrow supp\_intervals(p_i) \cup get\_support\_interval(m)$ /* Add newly covered interval */
        $next\_to\_check(p_i) \leftarrow get\_min\_end\_time(m)$
      end if
    end if
    $w_{start} \leftarrow w_{start} + 1$
  end for
end while
for $p_i \in \mathcal{PL}$ **do**
  $support(p_i) \leftarrow length(supp\_intervals(p_i))$
end for

The three remaining refinements – unification, class restriction, and instantiation– are also ordered, i.e., if a refinement has been applied it is not allowed to use any of the preceding refinement types any more. Within each of the refinements only variable positions after the the last refinement are allowed to be processed. For instance, if the third variable in a conjunctive pattern has been instantiated just the fourth or later variables can be used for further instantiation steps. Variables are only unified with one out of the set of preceeding variables. Unified variables are left out at the remaining refinement steps (as they are processed implicitely if the unified counterpart is restricted). At the class refinements the current class of a variable is specialized to one of its direct subclasses. Instantiations are only performed for variables which already refer to a leaf class in the class refinement. In all cases after a refinement as much information as possible is derived: after unification of two variables the class restriction at the corresponding positions is set to the more special class, after instantiation to instance o all variable positions in the class restriction are set to the corresponding class $i(o)$.

Due to space restrictions we can only sketch the proof for optimality of our refinement operator. For a complete proof the inverse refinement operators must be defined formally and it must be shown that no pattern is missed by creating the most special representation of a pattern after refinement. An optimal refinement operator is one that satisfies completeness and non-redundancy properties [Lee, 2006]. Analogical to Lee's proof it can be shown that these two properties hold; for more details see [Lee, 2006, p.97-99]. The inverse operations of the refinement operators $\rho^{-1}(p) = \rho_L^{-1}(p) \cup \rho_T^{-1}(p) \cup \rho_U^{-1}(p) \cup \rho_C^{-1}(p) \cup \rho_I^{-1}(p)$ are mutually exclusive as – depening on the counters in the pattern status $status(p)$ – just one of the inverse operations can be applied. Each of these operations itself leads to a single more general pattern which follows from their definitions (at $\rho_L^{-1}(p)$ just the last element can be removed, at $\rho_T^{-1}(p)$ the last restricted position is generalized to the set of all allowed temporal relations, at $\rho_U^{-1}(p)$ the mostright unified variable is split and replaced by a new variable, at $\rho_C^{-1}(p)$ the mostright restricted class is replaced by its single super class, at $\rho_I^{-1}(p)$ all occurences of the rightmost instantiated variable are replaced by a new variable). Assuming there exist two different paths (i.e., redundancy is given) from the most general empty pattern to a pattern $p \in \mathcal{L}_{MiTemP}$, $r_0 = \epsilon, r_1, \ldots, r_m = p$ and $s_0 = \epsilon, s_1, \ldots, s_n = p$ with $r_{i+1} = \rho(r_i)$ and $s_{i+1} = \rho(s_i)$. If the inverse refinement operator is applied to both r_m and s_n the resulting sequences must be identical due to the property of the inverse refinement operator with $r_n = s_n, r_{n-1} = s_{n-1}, \ldots r_1 = s_1, r_0 = s_0 = \epsilon$ and $m = n$ which contradicts the assumption of the two different paths. Thus, it follows that ρ is non-redundant.

```
directSubClassOf(team1,                holds(pass(p7,p6),27,29).
                 object).              holds(closerToGoal(p6, p7),23,31).
directSubClassOf(team2,                holds(uncovered(p6,p6),25,33).
                 object).              holds(uncovered(q9,q9),30,36).
directInstanceOf(p6, team1).           holds(closerToGoal(q8,q6),36,40).
directInstanceOf(p7, team1).           holds(pass(p9,p7),39,41).
directInstanceOf(p8, team1).           holds(closerToGoal(p7,p9),35,43).
directInstanceOf(p9, team1).           holds(uncovered(q8,q8),42,44).
directInstanceOf(q6, team2).           holds(uncovered(p7,p7),37,45).
directInstanceOf(q7, team2).           holds(pass(p8,p6),51,53).
directInstanceOf(q8, team2).           holds(closerToGoal(q7,q6),50,54).
directInstanceOf(q9, team2).           holds(closerToGoal(p6,p8),47,55).
holds(uncovered(q6,q6),12,14).         holds(uncovered(p6,p6),49,57).
holds(pass(p9,p8),15,17).              holds(pass(p8,p7),65,67).
holds(closerToGoal(p8,p9),11,19).      holds(uncovered(q6,q6),58,68).
holds(uncovered(p8,p8),13,21).         holds(closerToGoal(p7,p8),61,69).
holds(closerToGoal(q8,q9),16,26).      holds(uncovered(p7,p7),63,71).
```

Figure 2: Example input for evaluation

In order to show completeness it is necessary to proof that for each pattern $p \in \mathcal{L}_{MiTemP}$ a path $p_0 = \epsilon, p_1, \ldots, p_n = p$ exists. Here, again the inverse refinement operator and the status can be used. Let p be any pattern in $\mathcal{L}_{MiTemP}$. This pattern has the status $status(p)$ with the refinement level $n = |status(p)|$. If we get $p_{n-1} = \rho^{-1}(p_n)$ then $p_n \in \rho(p_{n-1})$ and $|status(p_{n-1})| + 1 = |status(p)|$. Referring to Lee we can find any p_i with $0 \leq i \leq n-1$ by applying $p_{i-1} = \rho^{-1}(p_i)$ and we know that $|status(p_i)| = i$ and $|status(p_0)| = 0$. By definition, the empty pattern is the only one with a refinement level of 0. Thus, we have found a sequence $p_0 = \epsilon, p_1, \ldots, p_n = p$ with $p_{i+1} \in \rho(p_i)$.

The candidate generation algorithm for lengthening differs from the other refinements as it is not applied to each pattern separately but to the set of frequent patterns of the previous step. The algorithm (Algorithm 2) is similar to *apriori-gen* [Agrawal and Srikant, 1994]. Starting from single predicate patterns in each following step patterns with the same $n-1$ prefix are combined in order to create new pattern candidates (cf. [Lee, 2006]). The difference here is that the "items" in the list are actually predicates which can appear multiple times in a conjunctive pattern. As the predicate order is also relevant for distinguishing the patterns no alphanumeric order can be used to just create one new candidate of two previous frequent patterns with identical prefix. Here, two patterns must be generated.

Algorithm 3 shows the support computation procedure. Input to the algorithm are the dynamic scene, the size of the sliding window, and the list of patterns to check. As long as the latest end time is not reached a window is moved over the sequence. At each window position just the "visible" predicates identified by the window position are taken into account for pattern matching. This has the advantage that during pattern matching many assignments do not need to be checked as they are out of range of the sliding window anyway. If a match is found for a pattern at the current window position the support interval list is extended by the support interval of the match and the next position to check

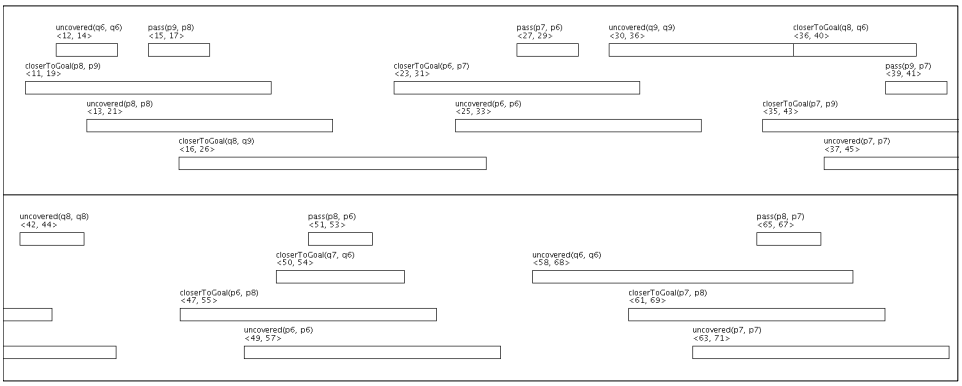

Figure 3: Test scenario

is assigned. If the match is valid beyond the window border some pattern matching steps can be omitted before the pattern has to be checked again. Finally, the window position is moved to the next position.

5 Learning 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 end time to every predicate. We developed a converter which automatically transfers a *MiTemP* input file to *ACE* input files. *ACE* is a data mining system which provides a number of different relational data mining algorithms including *WARMR* [Blockeel *et al.*, 2002; 2006]. Different problems had to be solved in order to set up *WARMR* to mine the same frequent patterns (with identical support calculation) as it is done by *MiTemP*. Due to space restrictions it is not possible to go into detail how the *WARMR* input is generated. A separate report capturing these details is currently written.

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. The transitive clauses for querying instances of classes and subclasses of a class can also be left unchanged and put into the background knowledge file. 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 predicate ID.

For setting up the learning bias in *WARMR* it is possible to define *rmode* statements. These statements define how a query can be extended during the generation of new query candidates. It is also possible to define constraints which must be satisfied in order to add an atom to the query. More details about the *rmodes* can be found, for instance, in Dehaspe's doctoral thesis and the *ACE* user's manual [Dehaspe, 1998; Blockeel *et al.*, 2006].

For each given *MiTemP* refinement as described in section 4 *rmodes* must be defined. For lengthening a *rmode* must be defined for each predicate definition. In order to avoid the same predicate instance being used more than once it must be guaranteed that the predicate ID variable differs from all other predicate ID variables of this query.

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 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.

Unification is handled by a special unification clause 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 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 arguments 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. 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.

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.

Setting up *WARMR* for computing the support as intended was a little bit trickier. *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. In combination with another predicate representing the window position (`inWindowPos`) for each temporal position it can be checked if a pattern holds.

Some tricks have made it possible to use *WARMR* as in-

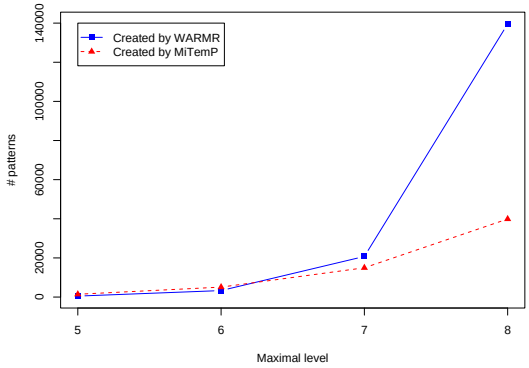

Figure 4: Number of created patterns

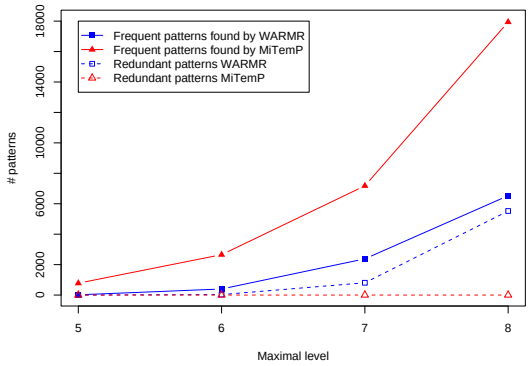

Figure 5: Number of frequent and redundant patterns

tended for mining temporal patterns. However, we had to accept some compromises in the solution. Computing the support is a little bit more inefficient as necessary as there is no way to use a real sliding window which covers an extended interval. In the current solution each time position has to be checked on its own (even if it can theoretically be known that the pattern holds at the current position due to the sliding window of an earlier position) and also predicate instances out of scope of the window might be checked.

Another problem is that redundant patterns are generated by *WARMR*. To the best of our knowledge it is not possible to avoid that for a unification two patterns are generated ($A = B$ and $B = A$). Furthermore, different representations can be generated for the same pattern if additional restrictions could be derived from the pattern (e.g., temporal relations using the composition table, class restrictions which should be specialized due to unification of variables as it is done by *MiTemP*).

In the case of *MiTemP* we required each pattern to be completely constraint w.r.t. temporal relations, i.e., that for each predicate pair exactly one temporal relation should be assigned. To the best of our knowledge it is not possible to define a constraint in *ACE* which guarantees to just create patterns which satisfy this property. This leads to the generation of some patterns which are out of scope of *MiTemP*.

Even though *WARMR* might have some drawbacks for our temporal pattern mining task it should be stated clearly that *WARMR* is not a special solution for mining temporal patterns from such an interval representation but a generic system for mining frequent queries which also can be used to mine queries representing temporal patterns.

6 Evaluation

The experiments with *WARMR* and *MiTemP* have different goals. First of all, it is a proof of concept that both approaches can be used to mine frequent temporal patterns. In order to find out if both approaches lead to the same support values the frequencies of all common patterns are compared. Furthermore, it is expected that constraining the search space at the refinement steps of the algorithm reduce the number of generated patterns a lot.

For the evaluation a simple soccer scenario has been used (Fig. 2). Different objects in the dynamic scene are objects `p6` - `p9` of class `team1` and objects `q6` - `q9` of class `team2`. Relations between these objects can be `uncovered`, `closerToGoal`, and `pass`. Fig. 3 shows the temporal validity intervals of the relations between the objects (time proceeding from left to right).

```
WARMR:
freq(8,96635,
     [currentIndex(A),getWindowPos(A,B),pass(C,D,E,F,B),
      uncovered(G,H,I,J,B),not(G=C),unif(I,H),unif(I,E),
      olderContemp(J,F),instanceOf(I,team1)],0.903614457831325).

MiTemP:
pattern(4544,
        [uncovered(_h6502661,_h6502661),
         pass(_h6502666,_h6502661)],temp(tr([olderContemp])),
         classRestr(team1,team1,object,team1),
         status(2,1,temp(1,2),2,varPos(2,2),0,varPos(-1,-1),
                1,varPos(1,1))) [Freq: 0.9036]
```

Figure 6: Example for a learned pattern

In different experiments the maximal refinement level of *WARMR* and *MiTemP* has been altered from five to eight. As *WARMR* could not create any complete pattern as defined above with a maximal refinement level below five these settings have been left out here. Table 2 summarizes the results of the test runs. Besides the number of created, redundant, and frequent patterns for both approaches it is also shown how many patterns have just been found by one approach up to this level and how many common patterns have been found by both approaches. The last two columns show the coverage, i.e., how many patterns of the other approach are covered at the current level. Fig. 4 shows a graph comparing the number of generated patterns at the different maximal refinement levels. Fig. 5 compares the number of mined frequent patterns and the number of redundant patterns of both approaches for the different levels.

All common patterns of *WARMR* and *MiTemP* get identical frequency values assigned. Fig. 6 shows example outputs of the same pattern by *WARMR* and *MiTemP*. While *WARMR* creates a number of redundant patterns (growing with an increasing maximal refinement level) the refinement operators in *MiTemP* are optimal as no redundant pattern was created (dotted lines in Fig. 5). *MiTemP* identifies much more frequent patterns at the different maximal refinement levels and creates less patterns at maximal refinement levels seven and eight– at levels five and six *MiTemP* creates more patterns. While the fraction of relevant frequent patterns to created patterns is quite low with *WARMR* (at level eight it is $\frac{6530}{139543} = 4.68\%$) almost every second pattern created by *MiTemP* is a relevant frequent one (at level eight: $\frac{17940}{39889} = 44.98\%$).

In the eighth level some patterns which have been mined by *WARMR* have not yet been found by *MiTemP* at this level. An inspection of the patterns has shown that these are patterns with many instantiations. Due to the refinement structure in *MiTemP* a variable is not instantiated before the class refinement restricts the variable to a leaf class (i.e., having no sub classes). In the *WARMR* solution in-

Max. level	#created *WARMR* patterns	#frequent *WARMR* patterns	#redundant *WARMR* patterns	#created *MiTemP* patterns	#frequent *MiTemP* patterns	#redundant *MiTemP* patterns	#common patterns *WARMR/ MiTemP*	#unique patterns *WARMR*	#unique pat-terns *MiTemP*	Coverage of *MiTemP* patterns in *WARMR*	Coverage of *WARMR* patterns in *MiTemP*
5	498	16	0	1436	779	0	16	0	763	$\frac{16}{779} = 2.05\%$	$\frac{16}{16} = 100.0\%$
6	3317	399	31	5087	2653	0	399	0	2254	$\frac{399}{2653} = 15.04\%$	$\frac{399}{399} = 100.0\%$
7	20754	2380	807	14962	7178	0	2380	0	4798	$\frac{2380}{7178} = 33.16\%$	$\frac{2380}{2380} = 100.0\%$
8	139543	6530	5523	39889	17940	0	6340	190	11600	$\frac{6340}{17940} = 35.34\%$	$\frac{6340}{6530} = 97.09\%$

Table 2: Results of the test runs with different maximal refinement levels

stantiation can be performed directly to a variable, i.e., the intermediate class refinement steps (specializing variables to `team1` or `team2`) are not needed and thus, some patterns can be created at an earlier refinement level.

7 Conclusion

In this paper we have presented an approach to temporal pattern mining which mines frequent patterns from time interval-based relational representations of dynamic scenes. An *Apriori* like algorithm has been introduced which performs a top-down search of the pattern space without multiple generation of patterns. The use of reasoning techniques creates the most specialized representation after refinement or identifies inconsistencies in patterns. This avoids the creation of "impossible" patterns which cannot be frequent as well as reduces the number of specialization steps which are implicit in the pattern already. Here, a composition table is used for identifying possible temporal refinements, and class information of variables is used to find the most special class of a variable by taking into account predicate definitions, variable unifications, and instantiations. An implementation of the well-known *WARMR* algorithm has been used to create another solution for the mining problem. The experiments have shown the advantage of avoiding the creation of impossible or redundant patterns in *MiTemP*. While less patterns had to be explored at refinement levels seven and eight much more frequent patterns have been found by *MiTemP* already. This can be particularly of importance if large sequences have to be processed, i.e., if support computation is costly.

Acknowledgment

We would like to thank Frank Höppner at the Fachhochschule Braunschweig/Wolfenbüttel for helpful discussions on support computations in temporal pattern mining. We also want to express our gratitude to the members of the DTAI research group of the KU Leuven, Belgium, for providing the *ACE* system including *WARMR* [Blockeel *et al.*, 2002]. Particularly, we would like to thank Jan Struyf for his great support with *ACE/WARMR*.

References

[Agrawal and Srikant, 1994] 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.

[Agrawal *et al.*, 1993] 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.

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

[Blockeel *et al.*, 2002] 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.

[Blockeel *et al.*, 2006] 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.

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

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

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

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

[Kaminka *et al.*, 2003] Gal Kaminka, Mehmet Fidanboylu, Allen Chang, and Manuela Veloso. Learning the sequential coordinated behavior of teams from observation. In Gal Kaminka, Pedro Lima, and Raul Rojas, editors, *RoboCup 2002: Robot Soccer World Cup VI, LNAI 2752*, pages 111–125, Fukuoka, Japan, 2003.

[Lattner *et al.*, 2006] Andreas D. Lattner, Andrea Miene, Ubbo Visser, and Otthein Herzog. Sequential pattern mining for situation and behavior prediction in simulated robotic soccer. In Bredenfeld et al., editor, *RoboCup-2005: Robot Soccer World Cup VIII*, pages 118–129. Springer Verlag, Berlin, 2006. LNCS 4020.

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

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

[Sagonas *et al.*, 2006] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, Luis de Castro, Rui F. Marques, Steve Dawson, and Michael Kifer. *The XSB System Version 3.0 - Volume 1: Programmer's Manual, Volume 2: Libraries, Interfaces, and Packages*, 2006.