



TECHNISCHE UNIVERSITÄT WIEN

DIPLOMARBEIT

Spatio-temporal patterns: Using spatio-temporal predicates to recognize situations in field sports

Ausgeführt am

INSTITUT FÜR GEOINFORMATION UND KARTOGRAPHIE (E127)

der technischen Universität Wien

unter der Anleitung von

O.Univ. Prof. Dipl.-Ing. Dr. techn. André Frank

durch

Armin Wasicek

Lederergasse 6, 4100 Ottensheim

Wien, February 21, 2006

Armin Wasicek

Danksagung

Für Ihre Unterstützung bei der Verwirklichung meiner Ideen, meinen Eltern.

Für Lob und Kritik an meiner Arbeit, meinem Betreuer Professor Frank.

Für Freundschaft und Zusammenarbeit, meinen Kollegen.

Für den Zusammenhalt, meinen Freunden.

Danke.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Time geography in sport	2
1.3. Connection to scientific discussion and outline	4
2. Data capturing techniques	5
2.1. Manual input	5
2.2. Video position extraction	6
2.3. Position tracking	6
2.4. Virtual sports	7
3. Spatio-temporal Data Types	8
3.1. ROSE Algebra	9
3.2. Temporal lifting operation	12
3.3. Abstract and discrete models	14
3.4. Moving object model	15
3.5. Tripod data model	17
4. Spatio-temporal Predicates	21
4.1. From spatial predicates to spatio-temporal predicates	22
4.1.1. Time dependent functions	22
4.1.2. Lifting predicates	23
4.1.3. Quantification	24
4.1.4. Evaluating predicates	26
4.1.5. Basic spatio-temporal predicates	26
4.1.6. Summary	26
4.2. Developments	27

4.3. Spatio-temporal Evolution based on the Tripod spatio-temporal data model	30
4.4. Recognition function	31
5. Case study: Rugby "Kick to touch" pattern	33
5.1. Modelling	34
5.2. Building predicates	37
5.2.1. Collect	37
5.2.2. Pass	39
5.2.3. Blocked	41
5.2.4. KickOut	44
5.3. Assembling the pattern	45
5.4. Summary	45
6. Implementing the concepts	47
6.1. Phases of specification and computation	47
6.1.1. Formalization of specification	47
6.1.2. Compilation of the model	48
6.1.3. Matching against a dataset	49
6.2. Theoretic concept to implementation in Prolog	49
6.2.1. The data	49
6.2.2. First stage: computation	50
6.2.3. Second stage: reassembly	51
6.3. Executing the prolog code	52
6.3.1. Calling goal: collect	53
6.3.2. Calling goal: pass	55
6.3.3. Calling goal: blocked	56
6.4. Summary and outlook	57
7. Conclusion	58
A. Acronyms	59
B. Bibliography	60
C. Sample implementation in Prolog	62

1. Introduction

This is about the interpretation of spatio-temporal data as chronology of sequential or parallel occurring situations. A situation is a dynamic incident, which incorporates participants and their interactions. Situation recognition is then the task to single out specific incidents from the space-time continuum. By comprising incidents in a situation, it is made comprehensive for human beings. In this work spatio-temporal predicates [6] are suggested to effect recognition of situations in field sports.

1.1. Motivation

The widespread use of modern position measuring technologies raises the need for new methods to deal with the collected data. Data inherently lacks the property to be well readable by us human beings. We need to find ways to facilitate the extraction of information out of raw data. This is a technical process called data mining or knowledge discovery. It uses statistical computations and pattern recognition to achieve its goal.

The use of information systems finds its way in sports. Computer scientists as well as sport scientists try to find possibilities to install new information technologies successfully to the domain of sport science. By revealing shortcomings or special abilities, these technologies empower the coaches to adjust the training methods more exactly to the athlete's specific needs. During a competition, the benefit of knowledge can change the odds.

In team sports the cooperation between single players is a crucial issue. It is difficult to evaluate cooperation, because individual perceptions lead to different interpretations of situations. Information systems can give a more objective view on the situation. For example, video judgement has become the final instance for a referee's decision in many sports.

Various techniques exist to measure positions of players in field sports. These systems can provide data rates exceeding 1000 measurements per second, accurate down to a few cen-

timetres. This represents a rather big data resource. To deal with this resource, data mining techniques have to be applied to find useful information.

1.2. Time geography in sport

In time geography ([12], [13]) Hägerstrand developed a set of visual tools for looking at geographic reality at the individual level. The motivation of his approach comes from the social sciences and the need to examine the spatial and temporal coordinates of human activity.

His model illustrates how a person moves through the spatio-temporal environment. Two horizontal axes form a two dimensional spatial plane. A third vertical axis represents time. His simplified world is enclosed in a cube, which he calls "aquarium" (Figure 1.1) and represents a portion of space-time.

Inside, lines represent the paths, which individuals follow through space and time. These so-called lifelines go from the bottom of the aquarium to the top in a continuous way. This corresponds to the limitation in human movement, that one can neither move in discrete steps, nor exist in more than one location at the same time. With Hägerstrand this is one of three limitations to space-time paths and called '*capability*'. The tool to visualize the capability to move is the *prism*, which represents the total area of space reachable by an individual. The shallower the slopes of the *prism*, the faster the individual can travel. A common sense term for this fact is the action radius, which makes an assumption, how far an object can move during a certain period of time.

Multiple paths, which link up temporarily, are called *bundles* and imply a possible interaction of people. This is '*coupling*'. In the sport context this happens for example when two players are fighting for the ball. A location, which remains unchanged for some time is called a *station*.

The last constraint is '*authority*' and refers to *domains*, which are areas with access limits. Note that the boundaries of these areas can change as well as time goes on, take for example the offside zone in soccer, which is expanding and contracting with the last defender's movement. In terms of Hägerstrand's time geography situation recognition is the locating of certain bundles in an aquarium. To identify the correct bundles the user has to enter some information about the situation prior to the recognition. Since an aquarium is a restriction to time and space, a bundle is itself an aquarium, which can again contain other aquaria. Thus a situation

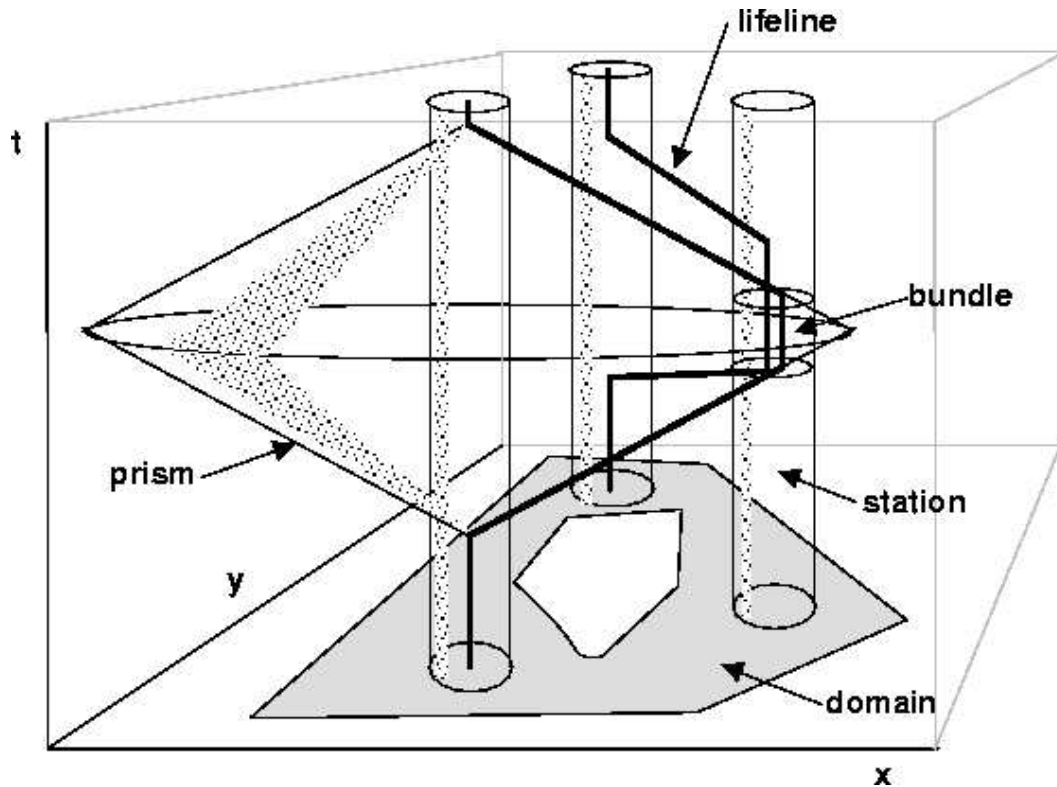


Figure 1.1.: Hägerstrand's aquarium: A space-time model

can be part of another situation and the recognition of a situation in a field sports match is the computation of a bundle in an aquarium.

Definition 1 (Situation recognition) *Situation recognition is the computation of a subset A' in a given situation environment A through a recognition function p .*

$$p : A \rightarrow A' \quad A, A' \dots \text{set of lifelines, } A' \subseteq A$$

Some fields of application make already use of situation recognition techniques; for example, collision detection systems for air traffic try to identify dangerous situations, or in road traffic, variable speed signs are installed to avoid traffic jams. There exist several reasons to introduce these techniques into the world of field sports. Studying spatio-temporal relationships in the sports domain has several advantages over the real-world social environment.

Following properties justify our interest to model them in a Geographic Information System (GIS) with regard to spatio-temporal properties:

- Bounded spatial and temporal extent (closed world assumption).
- Finite number of entities (pitch, players, ball, referee).
- Interaction between entities is well-defined.
- Simple rules apply to the game.
- Complex situations can occur during a match.
- Recurrent situations follow a certain pattern.

The first two properties constrain the world and its entities we are working with. This makes the computation more convenient for finite machines. The third and fourth property define in which ways the entities can be related temporarily or permanently. The fifth property says that nevertheless the elements of a field sports game are simple, they can develop a very complex structures. Finally the last one states, that for recurrent situations a fundamental pattern can be identified. An abstraction of this pattern can serve as a template.

1.3. Connection to scientific discussion and outline

We are living in a world of constant change. To understand the world, we got to understand this change. Studying time-varying concepts has a long history in philosophy, physics and computer science. Many models have been developed for different purposes and applications. They range from very simple to very complex like the theory of general relativity from Einstein. It has been shown that time and space are linked in many ways. So they have to be considered together.

In [4] an approach to spatio-temporal patterns is given. Erwig suggest the use of spatio-temporal predicates [6] to effect their recognition. The paper in [16] investigates possible ways to make use of different techniques from time geography in Rugby. The first two papers build the analytical foundation of this thesis, the third one describes how time geography can be useful in sports analysis.

2. Data capturing techniques

The goal of a data capturing method is to bring a set of incidents from physical reality into virtual reality. Then the virtual model can be subject to further processing. Various data collection methods exist to record the changing position of moving objects. The task performed by these techniques is to perpetually sample the location of a set of objects. Thus obtained positions are forwarded to a storage device and/or processed on the fly. The data collected in one of these ways represents the data source used in the following chapters.

This chapter explains the four most common methods to collect position data - manual input, video position extraction, position tracking and virtual sports.

2.1. Manual input

This is the low-tech approach to position recording: an observer logs the events from the pitch. This task can be performed simultaneous to the game (live) or from a data store (retrospective). TV companies use this technique to manually index their video recordings. This index helps with making new compilations out of existing video sources, like a match summary or "golden moments" at the end of the season.

However in our setup one observer is following the changing position of one or more players on a screen. On a recording device the observer is plotting the object's change of location. This procedure can be easily mapped to software: one window displays the video of the game, another one provides some sort of drawing program, where the objects of a game can be placed and moved on a virtual pitch. The actual position measurement is effected by the visual judgement of the observer. There are three possible sources for inaccuracies: First the observer's guess of the position; second the manual action of drawing; third the grid used by the software adds some inaccuracy.

- **Pro:** Easy setup: Just one computer is needed and the software is rather simple to use, is commercially available and can be put quickly into service.

- **Contra:** Data capture is very cumbersome and uses lots of manpower. The accuracy of the data is low.

Examples. The SCRUM (Spatio Chronological Rugby Union Model) developed by the Information Science Department, University of Otago [17] is a system to virtually move players over a pitch and watch a match video at the same time in order to record the position data.

2.2. Video position extraction

This technique computes positions by applying a series of image processing and pattern matching algorithms on several video frames. An object's position is calculated in relation to static elements in the picture, which are previously identified (calibration). Images from several views are necessary to overcome the warping effects of the camera's lenses. The first system of this kind was named LucentVision and brought into service for the ATP tour in 1998.

Video position tracking is a very promising approach, since the matches recorded and broadcasted by TV companies build a large video resource. An efficient technology could help in digitizing matches from the past. This method is related to video indexing techniques, which are an issue of active research ([19]). One of the strong points of this technology lies in its remote sensing concept, so the measurement does not influence its outcome.

- **Pro:** Videos build a large resource of past matches. No extra setup required: uses existing cameras from broadcasting companies.
- **Contra:** Every camera perspective has to be calibrated.

Examples. A description of the LucentVision system is found under <http://www.bell-labs.com/org/1133/Research/Visualinfosystems/>, last accessed in June 2005.

2.3. Position tracking

At present position measuring devices get more and more affordable and start to penetrate our everyday life. They are built in cars and cell phones to improve our navigation. Another application is the tracking of moving objects. The signal propagation delay between sender and receiver devices is used to measure the position. Unlike with Video position extraction

systems no intervisibility is necessary.

The range of setups is broad: systems using satellites moving in the orbit and mobile devices on the earth surface are construed for global usage, other ones using fixed stations to provide this service within fixed boundaries for local usage.

In the context of sports, the latter setup is more important. Microchips are attached to the ball, the player's and referee's feet and interconnected relay stations are built around the pitch. Additionally the pitch and the station must be surveyed with a high resolution in order to configure the stations properly.

- **Pro:** Very accurate and reliable service.
- **Contra:** Big effort setup. New hardware required.

Examples. Global systems operating from outer space are the american GPS, the russian GLONASS and the european Galileo. Local systems are the Cairos System, (<http://www.cairos.com> accessed August 2005) and LPM (<http://www.lpm-world.com/html/press/press.htm>, accessed June 2005).

2.4. Virtual sports

As the name suggests, virtual sport events are purely computational matches. One big segment is sport games, one other sport simulations. Sport games are primarily designed for entertainment. Recent publications amaze not only with love to the detail, but also with very natural game compartment. Sport simulations emerge from a more academic corner. They are used to study the behaviour of multi-agent simulation systems or to test artificial intelligence algorithms. Since the holding of virtual sports is based on computations, it is easy to derive valuable information like position data directly from the memory.

- **Pro:** Data is available in digitized form. The captured data is accurate up to one hundred percent.
- **Contra:** Works only in its respective virtual domain.

Examples. The RoboCup Simulation league is a pure virtual soccer league with four divisions in 2005.

3. Spatio-temporal Data Types

Data types provide structures for organising data in a database. Out in the database world there exist many approaches to deal with temporal and spatial data. In the past these two domains have been treated separately. During the development of concepts for the two distinct domains, footnotes were set, that these concepts could also be useful to the respective other domain. From these deliberations the domain of spatio-temporal databases evolved, with the goal to research data models which model the affinity between time and space sufficiently. A spatio-temporal database embodies spatial, temporal and spatio-temporal information.

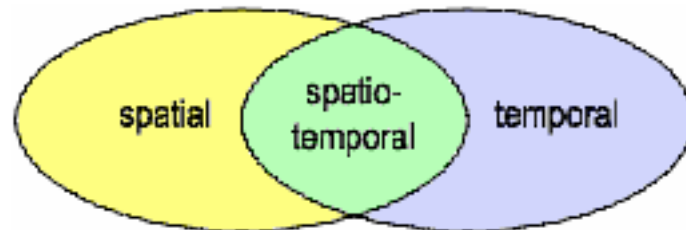


Figure 3.1.: *Spatial, temporal and spatio-temporal domain*

The special properties of spatio-temporal data raise the need for independent concepts for the spatio-temporal domain. The DOMINO project [18] surveyed following set of critical capabilities for spatio-temporal databases:

- **Location modelling:** Existing dbms cannot easily handle continuously changing data, like locations of moving objects. Updates of the location datum have to be performed constantly.
- **Linguistic Issues:** Traditional query languages such as SQL are inadequate for expressing queries. A database language must support spatial and temporal range queries.
- **Indexing:** For continuous change of location, the index has to be updated continuously, which sweeps off the benefit from indexing.

- **Uncertainty/Imprecision:** The object's location in the database cannot always be identical to the actual location. Some mechanism determining a measurement of the correctness of the answer must exist, e.g. the probability.

3.1. ROSE Algebra

An algebra for spatial data types is needed to allow queries to take into consideration the special nature of spatial data. Spatial Data Types (SDT) thus facilitate a conceptualisation of spatial entities in a database management system (DBMS). Basing the definition of these spatial data types on an abstract interface, separates the algebra from a particular data model of a DBMS.

ROSE stands for **RO**bstust **S**patial **E**xtension algebra. It realises a set of spatial data types and operations on them as extension to an existing DBMS. In this work it shows as a fundamental explanation of how spatial data types work and serves as basics for the following chapters, which continue with the concept of spatio-temporal predicates and are built upon the ROSE algebra. The ROSE algebra is defined together with a type system in [11]. In the following a short review is given.

To achieve its demand to be robust and have spatial extension capabilities, it puts up certain requirements:

- **Generality:** The geometric objects should be as general as possible, i. e. allow holes in regions; closed set of operations.
- **Rigorous definition:** Complete definition of carrier sets and operations to avoid ambiguities for programmer and user.
- **Finite resolution:** To be computable in finite devices.
- **Treatment of geometric consistency:** Distinct objects must be hold geometrically consistent in the database.
- **General object model interface:** An interface to an existing DBMS data model must exist. This is an implementation issue.

The ROSE algebra fulfills these criteria by using an underlying structure called a *realm*. Thus it defines realm-based spatial data types. A realm in this sense is a finite set of points and

non-intersecting line segments. Geometric objects on the realm are defined as elements of this set.

Realms are the foundation of the ROSE algebra. In contrast to euclidean space, which is a continuous plane, a realm got a finite representation. "A realm is a set of points and line segments over a discrete domain, that is, a grid, as shown in Figure 3.2" ([11]).

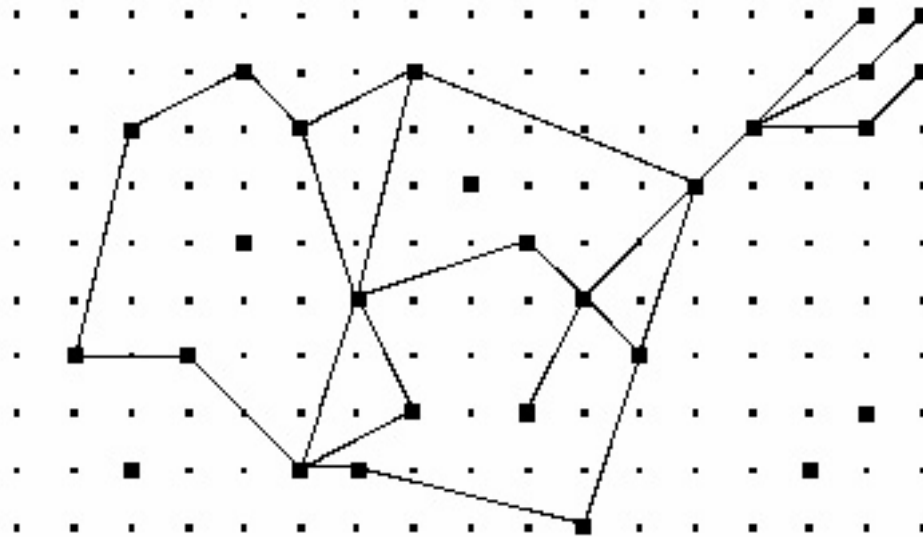


Figure 3.2.: *Example of a realm*

So a realm has two parts: a set of points (called *N-points*) and a set of lines (called *N-segments*), with a line consisting of a two point tuple. These are the basic elements of which the data types are composed. The two most important criteria to the algebra are, firstly that no point lies on a possible line segment and secondly no two lines intersect or overlap. This way every point of interest is contained in the underlying grid.

The problem arising in this context is, that intersection points of lines do not lie automatically on the grid and have to be transformed to fit into the realm. The rule is to move the real intersection point of two lines to the nearest point on the grid. Lines are to be transformed into "chains of segments". A so called envelope contains all the segments of a line. The number of line segments within the envelope changes due to the insertion and removal of intersecting lines. When a new line is added to the realm, new intersection points are computed and transformed. Existing lines are broken into two parts if necessary, to maintain the non-intersecting

property of the algebra. The envelope guarantees the error produced through this operation is minimized. The concept of a realm facilitates *generality* and *geometric consistency* as demanded in the requirements. One drawback is, that after performing several updates of the realm slight numerical errors may occur, due to the re-computation of intersection points.

The spatial data types of the ROSE algebra are forged in a layered approach. First a discrete space $N \times N$ is defined over $N = \{0, \dots, n-1\}$ a subset of the natural number set. This is the representation of the underlying grid. Only coordinates which are part of this space are valid. Computation is done in error-free integer arithmetic. This definition ensures some nice properties for the algebra and features a direct and robust implementation. Next layer are the *N-points* and *N-segments* - the two parts of a given realm over N . An *N-point* is a pair $(x, y) \in N \times N$. An *N-segment* is a pair of distinct *N-points* (p, q) [(p, q) and (q, p) are equal]. Some primitive predicates (with their natural language semantics) are defined to test the relation of two *N-segments*: meet, distinct, overlap.

Under this definition a realm can be viewed as a graph with the points as nodes and line segments as edges. Then we introduce the definition of a *R-cycle* as cycle and a *R-face* as face of this graph. A face is a cycle possibly enclosing other cycles, i. e. a region with holes. Furthermore a *R-unit* is a minimal *R-face*. These definitions facilitate the construction of a regions data type. Next a *R-block* is a collection of connected line segments and thus supports the definition of a lines data type.

The spatial data types of the ROSE algebra are:

- **points**: A point is an object, whose location but not extent is important. Described through a set of *R-points*.
- **lines**: A line is a connection in space. It has a starting and an ending point. Supported by a set of pairwise disjoint *R-blocks*.
- **regions**: An Object whose extent is relevant is called a region. It is defined through its boundaries. So a set of pairwise edge-disjoint *R-faces* is a region.

Some example operations on the data types of the ROSE algebra with their intuitive meaning:

- **Predicates**: equal, disjoint, inside, intersects, meets, ...
- **SDT operators**: contour, interior, plus, minus, ...

- **Number operators:** dist, diameter, length, ...
- **Set operators:** sum, closest, fushion, overlay, ...

Summary. The ROSE algebra defines realm-based spatial data types. It uses points and non-intersecting line segments to build the points, lines and regions data types. A realm is a grid and serves following purposes:

- Geometric consistency
- Closure properties
- No new intersection points are calculated.
- Realm data structure can be used as an index.
- Topological correctness precedes numerical correctness.

Most important, the ROSE algebra has an efficient implementation and is open to extensions.

3.2. Temporal lifting operation

The operation of temporal lifting has the function to add a temporal component to any entity. In the area of data types this function is realised as a type constructor. The application of type constructor τ on a given atomic data type α transforms it to the temporal data type $\tau(\alpha)$:

$$\tau(\alpha) = \underline{time} \rightarrow \alpha \quad (3.1)$$

If α is a spatial type, $\tau(\alpha)$ represents a mapping from time into space. For every timestamp on a time line time (*definition set*) exists a corresponding value in the *image set*. Thus a temporally lifted data type describes the domain and an attribute's changes over time.

In the *abstract moving object model* [8] time is considered to be linear and continuous. A graphical representation for this conceptualisation is the infinite time line, in algebraic terms we say time is isomorphic to the set of real numbers. Two structures are isomorphic, if they can be mapped onto each other and each part of one structure has a corresponding part in the other. For example the carrier set for temporal type instant is:

$$\underline{instant} := A_{instant} = \mathbf{R} \cup \{\perp\} \quad (3.2)$$

The moving type constructor performs the lifting operation similar to the τ operator. This term is used in the *abstract moving object model* to signify the newly gained possibility to record change (in location) of (spatial) data types. Here is the formal definition from [8]:

$$\underline{moving} := A_{moving(\alpha)} = \{f \mid f : A_{instant} \rightarrow A_\alpha, \text{partialfunction} \wedge \Gamma(f) \text{finite}\} \quad (3.3)$$

Now we obtain instead of the representation of a single value a partial finite function over time containing the changes of the value during a period of time as a data type. This function cannot be fully defined due to the fact, that the "recording" of the changing entity must have a start and an end. The latest value for end would be now. The finite property also makes the design implementable.

Since we want to work with values rather than with functions, we need a way to access the values in such a function. "The intime type constructor converts a given type α into a type that associates intime values with values of α ." By the use of this type single elements (instant-value-pairs) within such a function can be accessed.

$$\underline{intime}(\alpha) := A_{intime(\alpha)} = A_{instant} \times A_\alpha \quad (3.4)$$

Table 3.1 lists some generic operations taken from [8] to work with these data types. The argument types are on the left side of the arrow, the return type on the right side. The **at** operation extracts the value at a specific point in time. The extreme values are yielded by the **minvalue** and **maxvalue** operations. Note that a total order for type α must exist to be able to compute either of these operations. Next **start** and **stop** yield the respective values at some pre-defined point in time. For example **start** can mark a point when α changes from undefined to defined and **stop** when α changes from defined to undefined. The last four operations depend strongly on the definitions of type α and the notion of time time. The **duration** operation yields a real number representing a certain condition is valid. At last **const** performs the function of a wrapper for non-temporal types. Non-temporal types do not change over time and are thus constant. This operation serves as interface between temporal and non-temporal types.

$\tau(\alpha) \times \underline{time} \rightarrow \alpha$	at
$\tau(\alpha) \rightarrow \alpha$	minvalue, maxvalue
$\tau(\alpha) \rightarrow \underline{time}$	start, stop
$\tau(\alpha) \rightarrow \underline{real}$	duration
$\alpha \rightarrow \tau(\alpha)$	const

Table 3.1.: Generic operations for moving objects

3.3. Abstract and discrete models

Abstract and discrete models are two fashions of design and represent different levels of abstraction. Abstract data models use infinite sets, the temporal lifting operator τ produces types over an infinite domain. Remember time is defined continuous and infinite, i. e. isomorphic to the real numbers. For example consider a player running on a pitch; the player himself is modelled by a point moving through space. Viewed in 3D space (an aquarium, see section 1.2) the moving point morphs to a continuous curve. This continuity corresponds to physical reality. An example of objects moving discretely through time and space would be land parcels: an area changes its extent at a genuine point in time. When it comes to implementation, the continuous property is an insuperable barrier. Therefore the abstract model must be translated into a discrete one, which is computable. Thus the continuous curve representing the running line of a player morphs into a polyline, which is a set of connected line segments. Discrete models use finite representations of infinite sets. A discrete model is mostly an approximation of an abstract one and represents a subset of an abstract model's domain. One possibility to do the transition from continuous to discrete space-time is to divide the time line in slices like shown in Figure 3.3.

The design procedure of a data model is, first define an abstract model, next to derive a discrete one. Starting with a discrete model might lead to disregard some important options, which can be easily included in the abstract model. According to [5] (p10) there are two steps to develop a data model:

1. *Design a signature of a many-sorted algebra* ([14]): invent a number of names for types and operations between them. Formally a signature consists of *sorts* and *operators*, which are names for types and operations.
2. *Define semantics*: associate a many-sorted algebra by defining *carrier sets* for the *sorts* and functions for the operators. Carrier sets are collections of possible values for a *sort*

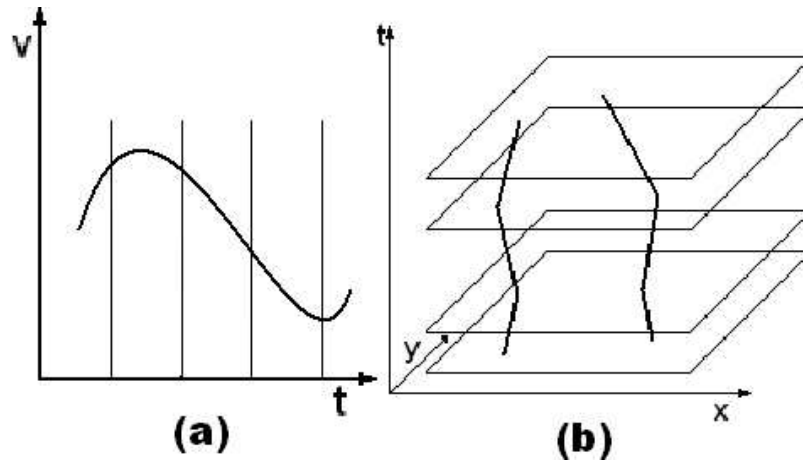


Figure 3.3.: Sliced representation of a moving real (a) and a moving point (b).

and functions are mappings between them.

Another possibility to model a continuous movement is to use a function rather than a set of observations.

3.4. Moving object model

This section gives a short summary of the *Moving Objects Model* defined in [8] as an example for an abstract spatio-temporal data model. It follows the abstract data type approach and defines a type system for spatial, temporal and spatio-temporal types and operations on them. Most important, it features temporally lifted types and operations.

Argument set	Type constructors	Instances
	\rightarrow BASE	<i>int, real, string, bool</i>
	\rightarrow SPATIAL	<i>point, points, line, region</i>
	\rightarrow TIME	<i>instant</i>
$\text{BASE} \cup \text{TIME}$	\rightarrow RANGE	<i>range</i>
$\text{BASE} \cup \text{SPATIAL}$	\rightarrow TEMPORAL	<i>intime, moving</i>

Table 3.2.: Signature of the type system of the abstract moving objects model

Basic types and type constructors. Table 3.2 shows the type system of the moving object model. There are type constructors with and without arguments. If a type constructor takes no

$$\begin{aligned}
A_{point} &= \mathbf{R}^2 \cup \{\perp\} \\
A_{points} &= \{P \subseteq \mathbf{R}^2 \mid P \text{ is finite}\} \\
A_{instant} &= \mathbf{R} \cup \{\perp\} \\
A_{moving}(\alpha) &= \{f \mid f : A_{instant} \rightarrow A_\alpha, \text{ partial function } \wedge \Gamma(f) \text{ finite}\} \\
A_{intime}(\alpha) &= A_{instant} \times A_\alpha \\
A_{range}(\alpha) &= \{X \subseteq \bar{A}_\alpha \mid \exists \text{ an } \alpha - \text{range } R(X = points(R))\}
\end{aligned}$$

Table 3.3.: Some carrier sets for the *Moving Object Model* data types.

arguments it is a type already and called *constant type*. Otherwise it can generate a new type out of itself and one instance of it's argument set. For example the *moving* type constructor applied to the constant *real* type yields the *moving(real)* type.

The base types have the common sense semantics and include an undefined value. The spatial types are similar to those of the ROSE algebra (see Section 3.1 or [11]). As said before, time is considered infinite and continuous. So the time type is isomorphic to the real numbers. A temporal type yields a mapping from time to the argument type. A range type contains subsets of types over totally ordered domains.

A total order on a domain is a binary relation, which sorts the elements according to antisymmetry (if $a \leq b$ and $b \leq a$ then $a = b$), transitivity (if $a \leq b$ and $b \leq c$ then $a \leq c$) and totality ($a \leq b$ or $b \leq a$).

Table 3.3 gives an overview of the carrier set for the types as defined in [10]. The *intime* type describes a value at a certain point in time. Under the assumption that a total ordering for the argument types exists, a range type can be defined to represent sets of intervals. Lines are interpreted as connected line segments (polyline), which are characterised by their endpoints and thus use the A_{points} carrier set. Note that due to the fact that sets can be infinite, every curve can be represented by an infinite point set. Regions interpret a point set for a polyline as graph (points as nodes, lines as edges). Then a region is a cyclic polyline. Finally a value of a *moving(region)* is a set of volumes in 3D space (x,y,t). An intersection of that volume with a plane parallel to the (x,y) plane yields the state of the region at specific time, a fact which is reflected by the *intime(region)* data type.

Operations. Different types of operations on the data types exist. Temporal lifting produces new operations out of existing operations through generalisation. Güting et al. identify a series

Operation class	Description	Example
Base type specific	Operations which are specific for some base types.	<i>And, Or, Not for Booleans.</i>
Predicates	Topological predicates.	<i>Inside, touches, attached, overlaps for point sets.</i>
Set operations	Fundamental operations on sets.	<i>Union, minus, intersection.</i>
Aggregation	Reduces sets of points to points.	<i>Minimum, maximum, average.</i>
Numeric properties	Structural computations.	<i>Counting adjacent areas.</i>
Lifting operations	Any argument of a non-temporal type is lifted to a temporal type, which returns a temporal type too.	The area hold by a football team.
Derivative	A measurement for the rate of change of an entity.	<i>Acceleration.</i>

Table 3.4.: Some operation classes of the *Moving Object Model*.

of operation classes (see Table 3.4) on the types. See [8] for more details including signatures of example operations.

Transition to discrete model. This breaks down the moving type constructor (of the abstract model) to a discrete mapping. One new type ('UNIT') and one new type constructor are introduced to replace the moving constructor. A 'UNIT' is a time interval - value pair, several UNITS are assembled in a mapping data type. Instead of a continuous partial function we got a set of time - value pairs.

The discrete representation of the moving object models introduces a new super-type called 'UNIT', which is a time interval - value pair. The mapping data type assembles a set of UNITS in order to make a linear approximation the continuous movement. A continuous curve is thus modelled by several snapshot values.

3.5. Tripod data model

This section reviews an example for a discrete data model. The origin of both the Tripod data model and the Moving Object Model 3.4 lie in the ROSE algebra 3.1. Both augment the spatial types to spatio-temporal types.

The Tripod System [9] implements a spatio-historical data model to represent spatial and aspatial data. It extends the ODMG standard [2] for object relational databases with the ROSE types, temporal types and allows reflection on an entity's evolution through the `historical` keyword. Spatial types are realised as 2D ROSE types, temporal types as 1D representation of them and the aspatial types are those from the ODMG standard. A historical attribute's changes caused by assignment operations are tracked in the database.

This database system is built on top of the PostgreSQL server. It features a thick layer 3.4 architecture approach. The first layer is an implementation of the ODMG standard, which specifies an ODBMS (object database management system). An ODBMS integrates relational or other non-object DBMS with object-oriented programming language capabilities and thus makes database objects appear as programming language objects.

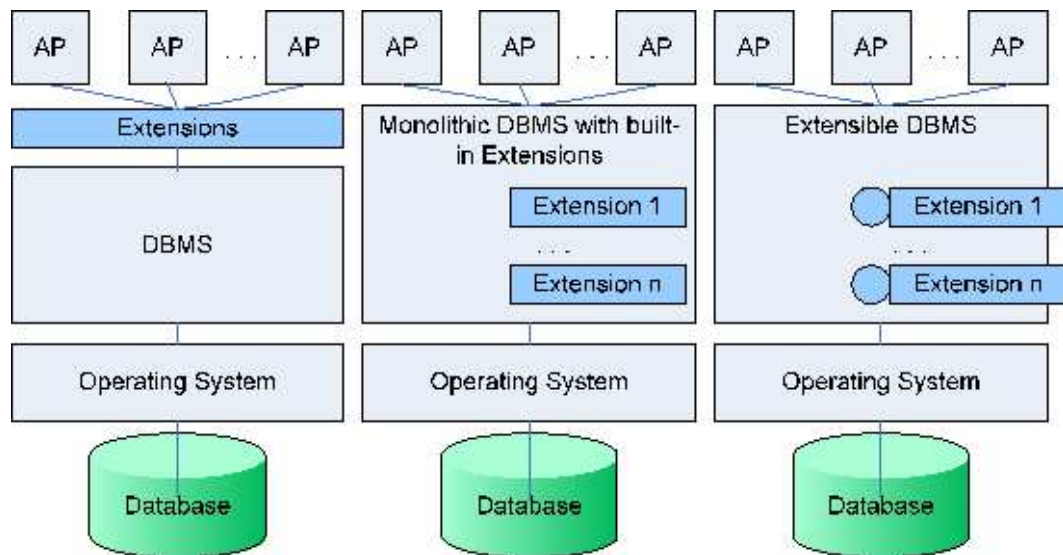


Figure 3.4.: *DBMS Extension architectures: (a) layered, (b) built-in and (c) data blade approach.*

"The key principles underpinning the Tripod project are *orthogonality* and *synergy*." The Tripod system adds spatial, temporal and historical functionality to a database. Orthogonality on the one hand means, that these features can be used separately in an effective manner, synergy on the other hand means, that the system allows a "combined use of spatial and temporal capabilities in a seamless and complementary manner". Shortly the new types and facilities integrate transparently in the existing system and can be used together or separately.

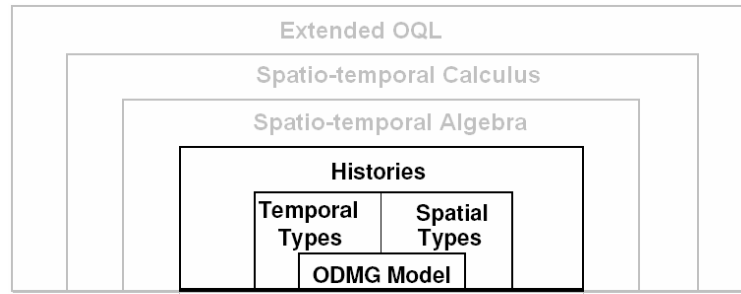


Figure 3.5.: Tripod architecture

Temporal extension. The Tripod system defines two types of timestamps, `Instants` and `TimeIntervals`. Either of them is a collection type, the first is holding instant values, the other half-open time intervals. Simple integers are used to represent instant timestamps. In addition every time type has a granularity γ . No more than one snapshot value can be stored for each granule of γ . This view of an underlying temporal grid with granularity γ corresponds directly to a realm in the ROSE algebra. Actually both temporal types are 1D projections of ROSE types. A subinterval of the `TimeIntervals` type uses of two instants to mark the begin and end of the period. Thus a series of intervals - $[t_i - t_j, \dots, t_p - t_q)$ - represents a `TimeIntervals` value; t_i and t_p are included in the time period, t_j and t_q not.

Spatial extension. The `Points`, `Lines` and `Regions` data types in the Tripod system are a direct implementation of the ROSE spatial types. They are set based and singletons of these types are used to access an individual element of a set. As discussed, a `Point` is a pair of coordinates in the underlying geometry, a `Line` a connection between two points and a `Region` a polygon. Note that spatial operations exist only on the set based types.

History mechanism. The goal of the Tripod history mechanism is to "provide functionality to support the storage, management and querying of entities that change over time." To achieve this, a history embodies information about the type (domain) of the tracked entity, a time model plus its granularity and a set of different states. These states represent an entities' changes due to assignment operations. So every left hand side construct of an assignment operation can have a history. The right hand side values collected in a history are called snapshots. So states are a set of time-value pairs, which are snapshots of the value at a certain point in time. They are related through an injective function:

$$states_H : \tau \rightarrow \sigma \quad \tau \in \mathbf{T}_H, \sigma \in \mathbf{V}_H \quad (3.5)$$

Every timestamp has a corresponding snapshot, but not every snapshot got a timestamp. These not related snapshots refer to non-historical types, which got the same value at all times.

Definition 2 (Lifeline, Aquarium) *The lifeline of an individual object is a history corresponding to [9]. An aquarium (according to [12], [13]) sets the boundaries to a collection of lifelines in time (starting time up to now) and space (domain \mathbf{V}_s of a spatial type)*

2.1	history:	$H = \{\mathbf{V}, \theta, \gamma, \Sigma\}$	\mathbf{V}	Domain of changing values
			θ	$\{\text{Instants}, \text{TimeIntervals}\}$
			γ	Granularity of θ
2.2	state:	$\langle \tau, \sigma \rangle \in \Sigma$	Σ	States collection of the form $\langle \tau, \sigma \rangle$
			τ	Timestamp
2.3	aquarium:	$A = H^*$	σ	Snapshot

In Definition 2, the history definition from [9], page 36 is cited: "A *history* is a quadrupel $H = \{\mathbf{V}, \theta, \gamma, \Sigma\}$, where V denotes the domain of values whose changes H records, θ is either `Instants` or `TimeIntervals`, γ the granularity of θ and Σ is a collection of pairs, called *states*, of the form $\langle \tau, \sigma \rangle$, where τ is a timestamp and σ is a snapshot."

4. Spatio-temporal Predicates

This kind of predicate logic addresses the change in spatial relationships over time. Starting with a small set of elementary predicates as "building blocks", the user is able to build more and more complex ones. Spatial relationships are defined as topological predicates. The 9-intersection model [15] provides a canonical collection of topological predicates for each combination of spatial types. In this context three basic spatial types - *points*, *lines* and *regions* - are identified. Examples for topological predicates are *meet*, *overlap* and *inside*. A spatio-temporal relationship is then "a sequence of (well-known) spatial relationships that hold over time intervals or at time points; we will call it a development" [6]. Spatio-temporal predicates then express spatio-temporal relationship facts which can be either true or false.

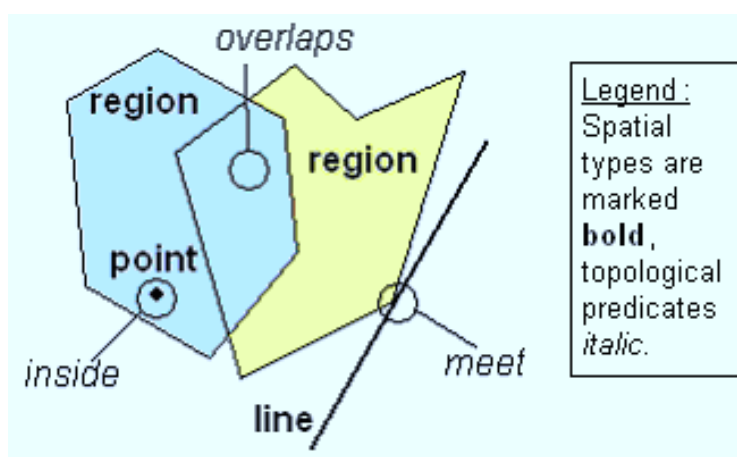


Figure 4.1.: *Spatial types and topological relationships*

The idea of spatio-temporal predicates is taken from [6]. Various other publications have small sections on spatio-temporal predicates ([4], [7], [8]). Similar work has been done by [3]. The rest of this paper explains the idea of temporal lifting, a powerful concept to integrate the temporal aspect in all kinds of entities, then it presents the definition of spatio-temporal predicates and discusses some of the properties. Next it is shown how to build complex predicates - developments - out of the set of basic predicates. At last the approach to

spatio-temporal evolution done by Griffiths et al. [9] is presented.

4.1. From spatial predicates to spatio-temporal predicates

Egenhofer defined in [15] a collection of meaningful spatial predicates, which represent topological relationships. See figure 4.1 for an illustrative example. Now we want to extend these predicates to the spatio-temporal case, in order to represent spatio-temporal relationship facts, which can be either true or false. This is achieved by the operation of temporal lifting in section 3.2. Spatial predicates can be seen as special case of spatio-temporal predicates. They are defined for certain points in time.

Consider a barn standing next to a road. The topological relation "barn near road" holds as long as both barn and road exist. If the barn burns down or the road decays, the quoted predicate will evaluate to false. To evaluate the "barn near road" predicate to true, two conditions must be true: both objects have to exist at the same time and the spatial relationship must hold. Thus a spatio-temporal predicate describes the change of a spatial (topological, geometrical) relationship. To enable common spatial predicates to fulfill this task, their temporal extent has to be extended from one instant to a time line. This is called temporal lifting. Analogies exist in all branches of science; many physical functions are dependent on time, sports people monitor a runner's heartbeat in a function over time, in biology different populations form and vanish at distinct points in time and space.

4.1.1. Time dependent functions

Another example for a time dependent function is the distance between two players during a match. As the players are in a constant movement, their distance changes over time. Time is considered to be linear and continuous, so for every instant a distance value between our two players exists. Following this definition, this value is a real number, distance over time a function:

$$\underline{distance} := \underline{position}(\alpha) \times \underline{position}(\beta) \rightarrow d \quad \alpha, \beta \in \mathbf{Player}, d \in \mathbb{R} \quad (4.1)$$

It reads: "The distance of two players α, β is computed as a cross product between their positions." This phrase states nothing about time, so whenever there are two positions, their distance can be computed. After adding the temporal component or making the distance

dependent on time, this distance value is replaced by a function over time. We can use the cross product notation as place holder for any binary operation.

$$\underline{distance}(t) := \underline{position}(\alpha, t) \times \underline{position}(\beta, t) \rightarrow d(t) \quad \alpha, \beta \in \mathbf{Player}, d(t) \in \mathbb{R} \quad (4.2)$$

The evaluation of $\underline{distance}(t_0)$ takes the positions of two players α, β at time point t_0 and yields the distance as a real number r . If $\underline{position}$ is not defined at one instant t_i on the time line for α, β , it returns the undefined value ' \perp '.

4.1.2. Lifting predicates

With spatio-temporal predicates we want to compute Boolean values rather than numbers and functions. It would be interesting to find out whether two players stand next to each other (meet) during a match. The spatial predicate 'meet' from the 9-intersection model [15] formalizes this relation. It is a function from two spatial objects to a Boolean value.

$$\underline{meet} := \alpha \times \beta \rightarrow b \quad \alpha, \beta \in \mathbf{A}, b \in \mathbb{B} \quad (4.3)$$

\mathbf{A} is a set of spatial object designators *point*, *region*, which we call data types. Güting et al. introduce in [8] the second order operator tau (τ): Given a set of arbitrary atomic data types \mathbf{A} , the application of type constructor τ on any element $\alpha \in \mathbf{A}$ lifts any flat data type α to a temporal data type. Thus tau (τ) effects the operation of temporal lifting, which is adding a temporal component to its argument. Formally:

$$\tau(\alpha) = \underline{time} \rightarrow \alpha \in \mathbf{A} \quad (4.4)$$

We can apply temporal lifting to the non-temporal (flat) types α, β :

$$\underline{meet} := \tau(\alpha) \times \tau(\beta) \rightarrow \tau(b) \quad \alpha, \beta \in \mathbf{A}, b \in \mathbb{B} \quad (4.5)$$

Similar to $\underline{distance}(t)$ defined in the last section, \underline{meet} describes now a function over time. Now we got a function to describe the changing relationship of two players. At some instants they meet, at some they don't. The function reflects this behavior. Evaluations of the function can be performed points in time specified as argument to the function. The behavior of a function at a certain instant is similar to the behavior of a corresponding spatial predicate.

Different evaluations at different points in time are likely to have distinct results. What we now want to do is to set those single evaluations in relation to each other in order to unlock the spatio-temporal information.

Note that temporal lifting yields a partial function. In places where it is not defined it returns the bottom element ' \perp ' and is thus extended to a total function.

4.1.3. Quantification

The nature of a predicate is to yield a single value, generally either true or false, not to yield a function. The magic word to solve this problem is quantification. In natural language, examples of quantifiers are for all, for some, many, few, a lot. Examples for phrases are: "Two players meet in a spatio-temporal sense, if they meet at least once." Or "Two players meet in a spatio-temporal sense, if they meet for all time." The first phrase is easy to understand, if there is one instant on the time line, for which the spatial relationship 'meet' holds, the spatio-temporal predicate 'meet' holds. The second uses another form of quantification and is a bit more difficult. The spatial predicate 'meet' has to hold for the two players for all time. Earlier we defined time as linear and continuous. So time has no beginning and no end. The conclusion is, in order to evaluate the spatio-temporal predicate 'meet' to true, the spatial relationship 'meet' must hold infinitely.

In the previous paragraph two possible quantifications were presented, in the examples namely the existence ('one instant exists', ' \exists ') and the unification ('for all time', ' \forall ') quantifiers. The assumption, that two objects share infinitely one relationship is highly unrealistic, so some restrictions are introduced. Quantifiers are restricted to perform evaluation only on a part of an object's lifetime. A lifetime in this context are the periods, where an object is defined. Different restriction levels exist, the default is, that a spatio-temporal predicate only operates on the common lifetime of its argument. In table 4.1 a summary of all different time notion modes is given.

A *temporal Boolean* is a function from time to the boolean set united with the undefined element, $\tau : \text{time} \rightarrow \{\text{true}, \text{false}, \perp\}$. This is the usual return type of a lifted predicate. In order to get a spatio-temporal predicate, which evaluates to a single Boolean, this intermediate result must be aggregated or quantified.

A binary spatio-temporal predicate is then a function from two temporal types to a single

Boolean value:

$$\tau(\alpha) \times \tau(\beta) \rightarrow \mathbb{B} \quad \alpha, \beta \in \{point, region\} \quad (4.6)$$

This concept can be easily extended to n-ary predicates, but in the following we will mainly deal with binary spatio-temporal predicates.

Table 4.1 presents five different ways to quantify a temporal Boolean returned by a binary spatio-temporal predicate. The leftmost column says how the two temporal domains of the two argument objects are set in relation, additionally a description and a diagram are provided. The first quantification policy is the most restrictive, the last one is the least restrictive. The additional δ^{th} property works on time instants. The two columns outside right are a notation to mark the predicates with their corresponding quantification policy and its semantics. Quantifiers are the existence (\exists) and the universal (\forall) quantifiers. The semantics of the first one is, if there exists at least one point in time, where the predicate expression is true for both objects, the predicate evaluates to true. The for all quantifier means, that the temporal Boolean must be true for the whole time span yielded by quantifying the objects' temporal domains. The return value of a quantified temporal Boolean is a single boolean.

Time	Description	Picture	Short	
$time$	both objects hold over all times - unrealistic			
$t_1 \cup t_2$	both arguments have the same lifetime		\overleftarrow{p}	$\forall_{\cup} p$
$t_1 \subseteq t_2$	one is part of the other's lifetime		\overleftarrow{p}	$\forall_{\pi_1} p$
$t_1 \supseteq t_2$	one is part of the other's lifetime		\overrightarrow{p}	$\forall_{\pi_2} p$
$t_1 \cap t_2$	holds on common lifetime		\bar{p}	$\forall_{\cap} p$
$t_1 \in t_2$	instant on lifetime		\dot{p}	$\exists p$

Table 4.1.: Time quantification policies

4.1.4. Evaluating predicates

Following steps are to be performed for evaluating a spatio-temporal predicate $\Theta_\gamma p$ with a quantifier $\Theta \in \{\forall, \exists\}$ and a quantification method $\gamma \in \{\cup, \cap, \pi_1, \pi_2\}$.

1. The temporal domain \mathbf{T} on which both predicates operate is computed:

$$\mathbf{T} \leftarrow \gamma(\text{dom}(S_1), \text{dom}(S_2))$$

2. The predicate is aggregated for the whole temporal domain in a temporal Boolean:

$$\mathbf{B} \leftarrow p(S_1(t), S_2(t)) \forall t \in \mathbf{T}$$

3. The temporal Boolean is quantified:

$$\text{if } \Theta b_i \in \mathbf{B} = \text{true} \text{ then } \text{true} \text{ else } \text{false} \quad \text{with } \Theta \in \{\forall, \exists\}, 1 \leq b_i \leq |B|$$

The next section presents several relations between temporal data types in more detail (Section 4.3).

4.1.5. Basic spatio-temporal predicates

Basic spatio-temporal predicates are the spatial predicates from the 9-intersection lifted to spatio-temporal predicates. For each of those predicates exists a default temporal aggregation which corresponds to the common sense interpretation. In the literature [6] this is explained for the predicate *disjoint* in the following way: "For example, when we ask by using a *disjoint* predicate whether the route of a plane did not encounter a storm, we usually require disjointness only on the common lifetime, that is, the result of this query is not affected by the fact that either the storm or the flight started or ended before the respective other object. Thus, the preferred of default interpretation for spatio-temporal *disjoint* is the predicate $\overline{\text{disjoint}}$."

Two sorts of predicates can be distinguished, instant and period predicates. The former hold on one point in time and start with a lowercase letter, the latter hold for a timespan and are indicated by an uppercase initial letter, to be in agreement with the literature. In Table 4.2 a choice of default aggregations for some basic spatio temporal predicates for regions.

4.1.6. Summary

Summarized, three steps have to be taken to create a basic spatio-temporal predicate:

- Take a spatial predicate.

Disjoint	$:= \overleftarrow{\overrightarrow{disjoint}}$
Meet	$:= \overleftarrow{\overrightarrow{meet}}$
Overlap	$:= \overleftarrow{\overrightarrow{overlap}}$
Equal	$:= \overleftarrow{\overrightarrow{equal}}$
Covers	$:= \overleftarrow{\overrightarrow{covers}}$
CoveredBy	$:= \overleftarrow{\overrightarrow{coveredBy}}$
Contains	$:= \overleftarrow{\overrightarrow{contains}}$
Inside	$:= \overleftarrow{\overrightarrow{inside}}$

Table 4.2.: Some default aggregations ([4], [8]).

- Lift it to a spatio-temporal function to add the temporal component.
- Reduce function to a spatio-temporal predicate by quantification.

Two major classes of predicates can be distinguished by regarding their temporal aspect: instant predicates and period predicates. The latter can hold only for a period of time, whereas the former can hold for an instant in time as well as for a period of time. This property is due to the fact that they are defined in a continuous context. In the following lowercase letters imply an instant predicate and uppercase letters a period predicate.

Alternatively, a two-dimensional spatial object moving along the time axis can be considered as a three-dimensional object. In this sense a point becomes a line, a line a region and a region a volume. Then the topological relations between three-dimensional objects can be used as spatio-temporal predicates. For every basic spatio-temporal predicate exists a corresponding three-dimensional predicate, except for *Disjoint*, *Inside* and *Contains*.

4.2. Developments

Several spatio-temporal predicates can be combined to build a more complex predicate. Temporal combinators serve as glue between the predicates.

Developments are sequences of spatio-temporal predicates. By means of temporal combinators, several predicates are connected to build a more complex predicate. Single predicates are evaluated one by one in a consecutive order. If all the predicates and their composition holds, the development holds. So every spatio-temporal predicate can possibly consist of other predi-



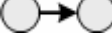
Compositor	Syntax	Description	Automaton
From	$p \vdash P$	P holds from a certain point in time.	
Until	$P \dashv p$	P holds until a certain point in time.	
Then	$P \dashv p \vdash Q$ $P \triangleright Q$	P holds until a certain point in time; then Q holds.	

Table 4.3.: Temporal combinators

cates and thus be a development. This conception enforces the reuse of existing developments for new purposes.

Predicate combinators. Predicate composition defines three possible operations: *from* - *until* - *then*, with the same temporal semantics like in natural language. Table 4.2 gives a detailed overview of the semantics of these operators. The syntactical and the language description are found in [6]. Informally, combinators wire single predicates together. Temporal composition is not the only possible way to form a development. A more advanced combinator is the alternative, which branches the timeline into several possible ways to follow, or the reflection - the development is executed in reverse order after reaching its end.

Almost any logical formula can be interpreted as an automaton. The rightmost column in Table 4.2 shows parts of automons to assign them the spatio-temporal meaning of the temporal combinators. Here an explanation based on automata theory:

Automata. An automaton is a mathematical model for a finite state machine (FSM). A FSM reflects changes of the model driven by the input data from its starting time to the present moment. Automata consist of states and transitions. Given an input the FSM jumps through a series of states following its transition function. Thus an automaton stays in one state until a transition function triggers the next state.

What we want to do is to use automaton interpretations of spatio-temporal developments to visualize and facilitate their composition. In the diagrams in Table 4.2, states are to be taken for period predicates and transitions for instant predicates. Automata provide a possible visualisation for spatio-temporal developments like discussed in [7]. See Figure 4.2 for example,

where the events forming a pass are formulated as a development and illustrated in an automaton. The next section explains in more detail how to describe or specify situations with spatio-temporal predicates.

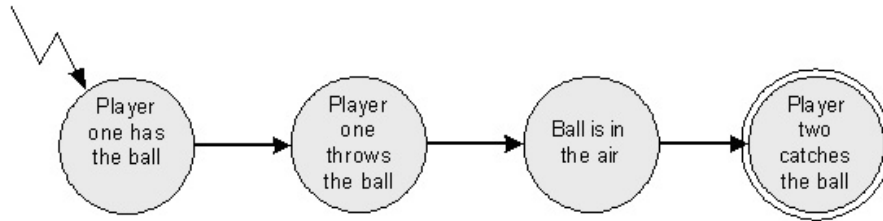


Figure 4.2.: *An automaton for a pass*

In a development the order of predicates cannot be arbitrary, since objects do not jump discretely through time and space, but move continuously. Figure 4.3 summarizes the predicates listed in Section 4.1.5 in a so called development graph. Such a graph provides possible transitions between predicates like in an automaton. Another interpretation is the development graph as an automaton for spatio-temporal objects. Changes in state of the object are allowed only along the edges.

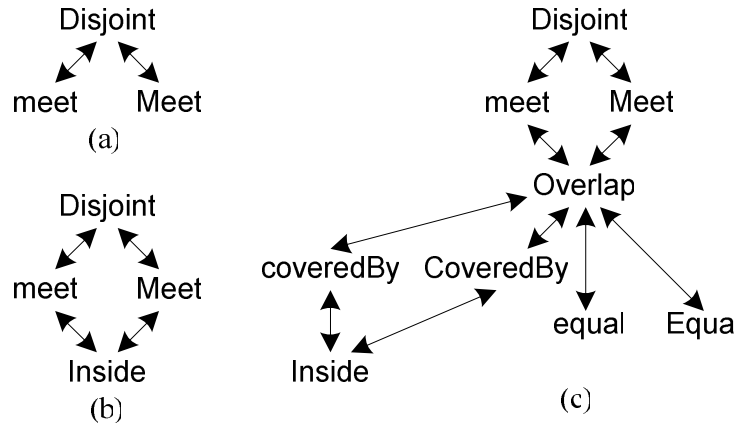


Figure 4.3.: *Development graph for (a) points, (b) lines and (c) regions*

4.3. Spatio-temporal Evolution based on the Tripod spatio-temporal data model

The paper [3] contributes an approach to spatio-temporal queries. In this context queries involving an object's change over time are called *evolution queries*. The algorithms presented in the paper work on the historical data types of the Tripod data model (see Section 3.5, [9]). They provide the functionality of a "spatio-temporal algebra" as shown in the graphical description of the Tripod implementation in Figure 3.5.

A *change pattern* is a "chronologically ordered sequence of observations". These observations can be made upon a single entity's history or multiple entities' relationships. It is supposed that an entity or the relationship changes its state over time and is supposed to follow a change pattern. This automaton behavior is similar to spatio-temporal developments described in Section 4.2.

To make a change pattern computable, it is specified as regular expression over the interpretation structure. This structure takes the form of a list with the elements $\{t, f, u\}$. The elements are arranged in temporal order and each element states the change in value at a certain granule. In this context t stands for true (there is a change), f for false (no change) and u is the undefined element. Note the similarity to the definition of a temporal Boolean in Section 4.1.3. Various mapping operations can precede the building of this structure. For example to query if the distance between two objects changed during a period of time. First the distance function builds a list of distance values for two objects p_1 and p_2 , next two subsequent list elements are checked for difference, which results in the intermediate representation:

Given two lists p_1, p_2 with:

$$\begin{aligned}
 p_1 &= \langle (3, 1), (3, 0), (2, 3), (2, 4) \rangle \\
 p_2 &= \langle (1, 1), (3, 2), (2, 2), (2, 1) \rangle \\
 d_0 &\leftarrow \underline{distance}(p_1, p_2) = \langle 2, 2, 1, 3 \rangle \\
 \underline{distchange}(d_0) &= \langle f, t, t \rangle
 \end{aligned} \tag{4.7}$$

The *distchange* list can now be aggregated with the for all quantifier (i.e. the distance does not remain equal) resulting in false, or the exists quantifier resulting in true (i.e. there is a change in distance).

Summary. Following steps have to be performed to evaluate an evolution query in the Tripod system:

- Apply mapping if necessary.
- Build intermediate representation.
- Evaluate regular expression.

Three kinds of possible queries are presented in [3], namely intra-history cross-timestamp (IHC), cross-history cross-timestamp (CHC) and cross-history intra-timestamp (CHI). The example above performs a IHC type query. Starting with two histories one intermediate list is generated and the elements of the list are set in relation to each other.

- IHC Evolution query over consecutive snapshots within a single (non-empty) history
- CHC Evolution query over consecutive snapshots in a paired (non-empty) history
- CHI Evolution query over snapshots in a paired (non-empty) history at each timestamp

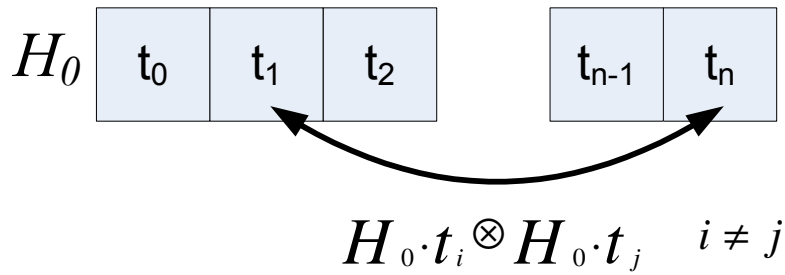


Figure 4.4.: *IHC evolution query over consecutive snapshots within a single (non-empty) history*

4.4. Recognition function

Now that we know about spatio-temporal predicates, we can explain how to use them for situation recognition. Following Definition 3 a recognition function is constructed as a term of spatio-temporal predicates combined by compositors.

Definition 3 (Recognition function) *The recognition function is composed by one or more predicates through combinatoria.*

$$p = p_1 \circ_1 \dots \circ_k p_n$$

$$\circ_j \in \{\text{compositors}\} \quad 1 \leq j \leq k$$

$$p_i \in \mathbf{P} \quad 1 \leq i \leq n \quad \mathbf{P} \dots \text{set of spatio-temporal predicates}$$

Note that the compositora have to be used in a meaningful way.

A situation is broken down to activities, which are translated to appropriate predicates. Each predicate p_i stands for a single activity. They are put together again in a recognition function, which is similar to a spatio-temporal development or evolution query. Predicate compositora serve as glue between the predicates. They structure the temporal succession of activities and thus have to be chosen in a meaningful and unambiguous way.

The recognition function thus provides a genuine formal description of a situation and is computeable, because its fundamental functions and compositora are computeable. A function can be applied to a spatio-temporal dataset. Following Definition 1 it yields a subset of the input data. Thus it performs the operation of a filter. The result has to be interpreted with respect to the information, which has to be gained. Examples are the counting of the occurrences of a situation, or length of a player's route. The next Chapter 5 shows the case study of how to translate a common game situation to a recognition function.

5. Case study: Rugby "Kick to touch" pattern

Game patterns represent an abstraction of moves trainers draw on, say, a blackboard or whiteboard to show their players how to field the team. They consist of a basic initial position configuration and possible further moves. In Figure 5.1 an example for such a drawing is given. It demonstrates the very common 'Kick to touch' game pattern in rugby.

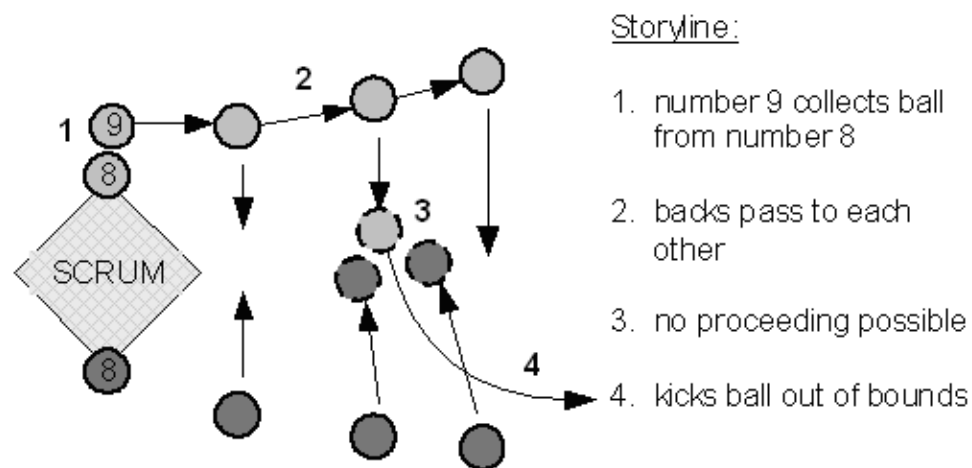


Figure 5.1.: Rugby "Kick to touch" pattern drawing

Consider the picture in Figure 5.1. The starting position on the pitch is a "scrum" (in this paragraph, rugby specific words are in quotes), which is a formation of players set by the referee. Up to eight fellow team mates hold on tight to each other to be able to act as one pushing machine. After the referee blows his whistle to restart the game, the ball is deployed into the "scrum". Now the opponent "scrum-halves" are pushing each other to gain ball possession. Eventually the ball leaves the back of the "scrum" via player number eight ("No.8"). This action marks the start of the game pattern:

1. Player number nine ("Scrum-Half") collects the ball from player number eight's feet.

2. A series of passes among a formation of players, called the "backs" follows. At some stage no forward progress on the pitch is possible.
3. Mostly this happens, when the defending team stops the player in possession of the ball.
4. Sensing this and to avoid a possible loss of ball possession, the player holding the ball kicks it out of bounds.

Figure 5.2 is an abstraction of this game pattern in the form of an automaton. An automaton or finite state machine (FSM) consists of states and transitions, which are noted as circles and (directed) arrows. Only one state can be active and external events trigger transitions, which are changes of the active state. Possible events can be everything from reading a character from the input stream to passing a ball. An initial event puts the automaton into work. In the diagram the initial state is marked by a lightning arrow. An automaton can have only one initial state. When the automaton reaches a final state - noted by a doubly lined circle - it terminates. Automata which come to an end are called determined finite state machines.

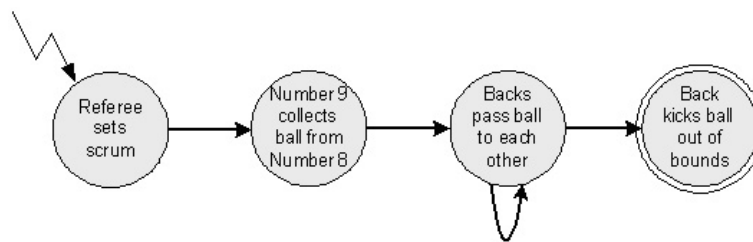


Figure 5.2.: Rugby 'Kick to touch' automaton (FSM)

5.1. Modelling

We now want to use a more precise description than pictures and natural language for the pattern. In [7] three categories of entities for describing a video scene are identified. We use the same definitions for our purpose (Table 5.1)

A look at the picture gives us the objects acting in the rugby video scene. There are several figures and one ball on a pitch. Figures can act as individuals or be summed up in a group (like a 'scrum' in the example). The player figures belong to teams, where they are distinguished by numbers. A team is divided into two groups, the forwards and the backs. An extra figure is the referee ("third team"). A pitch is characterized through its boundaries and playing space.

Entity	Description	In natural language	In the drawing
Object	Type of actors	noun, role	Shape
Activity	Type of action	verb + roles	Arrows
Event	Instance of activity	complete phrase	group of shapes and arrows

Table 5.1.: Actions and actors on the pitch

Here the objects from the example are expressed in an informal list notation with nested lists.

```

objects := { figures; ball; pitch }
figures := { players; referee }
ball := { ball }

pitch := { locations; boundaries }
players := { team_red; team_blue }

team_red := { forwards; backs }
team_blue := { forwards; backs }

forwards := { player_1 ... player_8 }
backs := { player_9 ... player_15 }

player_1 := { player_number_1 }
...
player_15 := { player_number_15 }

locations := { point; line; region }

point := { (x,y) ∈ Γ ⊂ ℝ2 }
line := { pi ∈ points | 2 < i < n, p0 ≠ pn, i ∈ ℕ }
region := { pi ∈ points | 3 < i < n, p0 = pn, i ∈ ℕ }

boundaries := { outline, far22line, near22line, ... }
outline := { r ∈ region | n = 4 }

```

```
setpiece := { lineout, scrummage }
```

```
scrummage := {  $p_i \in \text{team\_red} \vee \text{team\_blue} \mid 1 < i < 8$  }
```

```
lineout := {  $p_i \in \text{team\_red}, q_j \in \text{team\_blue} \mid i, j > 2 \wedge i = j$  }
```

Activities describe the dynamics of the system. They describe how and where the game flows. Objects take roles in activities, so activities describe scenes in a generic way. Instantiating an activity through filling the roles with objects yields an event. Following the storyline in Figure 5.1, we identify the activities in the example:

1. *[role player]* collects *[role ball]* from *[role player]*
2. *[role player]* passes to *[role player]* passes to ...
3. *[role player]* cannot proceed = is blocked by the other *[role team]*
4. *[role player]* kicks *[role ball]* to *[role boundaries]*

Finally we add the precondition: *[role referee]* sets *[role setpiece]* at *[role location]*.

A role says how someone or something can participate in an activity. This makes sure things don't get confused in the model. The word 'roles' is used as an intuitive synonym for data types. Next we need to determine which objects can act in which roles:

role player	→	players
role team	→	team.red team.blue
role ball	→	ball
role referee	→	referee
role setpiece	→	setpieces
role location	→	locations

Till now we've defined some activities and their participants, a signature in other words. It's time to give the signature of our activities some semantics by means of tying it up to spatio-temporal predicates. For our purpose the 5th quantification over time (see Table 4.1) is appropriate. It holds on the common lifetime for two objects and is the default for spatio-temporal predicates. Facts like one player leaves the pitch (due to an injury) the end of a player's "lifetime" - and gets replaced by a reserve player - the beginning of a "lifetime" - are thus sufficiently modelled.

Objects in a game are modelled as moving points. This is not the only possible geometrical interpretation of a match's structure. Formations like a scrum in rugby can be perceived as a moving region. The scrum region can be described as minimum bounding polygon over the set of players participating in the scrum. In [16] a whole team is understood as a polygon (region), which gives an interesting views on how two teams interact.

The spatial representation changes with the observer's perspective. The atoms are moving points (players, ball, referee) and depending on the context, which formation is executed, the observer's spatial perception changes. Single players are united to a formation with a certain name (e.g. scrum) and a certain spatial representation (e.g. a scrum is a region). The thus emerging zooming effect is quite natural; when many different things happen around us at the same time, the human brain tries to build abstractions and groupings, in order to keep the overview.

In this work we want to model players and ball as point entities. Furthermore, an ideal world is assumed, where the moving points meet each other in time and space *exactly*, when their respective entities interact directly. This is for example when a player catches the ball, then the moving points of the player and the ball meet. In the real world this cannot be true, because entities have an extent, a property which points don't have. Additionally, entities don't meet exactly, they touch in some way. This problem is solved by using an underlying grid (see Section 3.1). This ensures that points can meet exactly. Moreover measuring errors, etc. have to be taken into account. So real world data has to be corrected in some way.

5.2. Building predicates

5.2.1. Collect

Firstly, the activity 1 "[role player] collects [role ball] from [role player]" is considered. Collecting a ball is to grab it either from the ground, in the air or from another player's possession. Thus first one player must initially possess the ball. Next someone else nearby takes the ball. So the problem to solve is to model the two new activities - possession and grabbing - with spatio-temporal techniques:

1. [role player] possesses [role ball]
2. [role player] grabs [role ball]

A player can possess a ball by holding it in his hands. We can perform a check for every instant of the game, whether the player holds the ball or not. The partly aggregation of the resulting Boolean vector tells us for an interval of time, if the player possesses the ball or not (the mode of aggregation leaves space for an interpretation of 'possession' in rugby). The spatial translation of 'holding' is both player and ball share the same position (under the assumption we made at the beginning). So the mapping between activity 'possess' and a spatio-temporal predicate is:

$$[role\ player] \text{ possesses } [role\ ball] \rightarrow \quad Meet(p, b) \quad (5.1)$$

$$\exists p \in \mathbf{P}, \exists b \in \mathbf{B}$$

There is exactly one player out of all players, who can possess the ball. Grabbing describes the transition of bringing the ball in one's possession. So first the grabbing player is approaching the player in possession (spatially speaking all three are sharing the same space), followed by taking the ball and leaving the player without the ball. There is no spatial equivalent for taking a ball (at least not in 2D space).

$$[role\ player] \text{ grabs } [role\ ball] \rightarrow \quad Meet(p_1, p_2, b) \dashv meet(p_2, b) \quad (5.2)$$

$$\exists (p_1, p_2) \in \mathbf{P}, p_1 \neq p_2; \exists b \in \mathbf{B}$$

We model this in a 'from' predicate: "From the time of meeting on, the other player got the ball". Now we can put everything together, in the fashion of spatio-temporal developments:

$$Collect(p_1, p_2, b) \rightarrow Possess(p_1, b) \triangleright Grab(p_1, p_2, b) \triangleright Possess(p_2, b) \quad (5.3)$$

The picture in Figure 5.3 describes the three phases of the event $Collect(p_1, p_2, p_3)$. For each phase the corresponding predicate holds. Parallel trajectories indicate that the object meet each other in time and space.

Repeated application of the $C(\cdot)$ operator deducts the development $\Pi = Collect(p_1, p_2, p_3)$ by substitution to a sequence of spatio-temporal predicates:

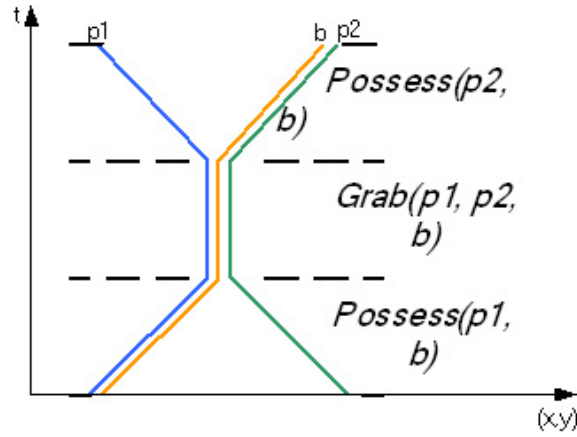


Figure 5.3.: Collect

$$\begin{aligned}
C(\Pi) &= C(Possess(p_1, b) \triangleright Grab(p_1, p_2, b) \triangleright Possess(p_2, b)) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash C(Grab \triangleright i \triangleright Possess) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash Meet(p_1, p_2, b) \dashv meet(p_2, b) \vdash C(Possess) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash Meet(p_1, p_2, b) \dashv meet(p_2, b) \vdash Meet(p_2, b) \\
&\quad \exists(p_1, p_2) \in P, p_1 \neq p_2; \exists b \in B \\
&qed. \tag{5.4}
\end{aligned}$$

Summary. Firstly an event was defined as an instance of an activity. An event exists, when its corresponding activity can be satisfied on a dataset. The entry and the exit point of the predicate determine the beginning and end of the event. For example, to count the occurrences of a game pattern is to find all the events of an activity in a dataset.

5.2.2. Pass

The next activity (2) "[role player] passes to [role player] passes to ..." introduces the concept of repetition. When a the execution of a predicate has ended, it begins again right from the start. This can be repeated for several times. This behavior corresponds in an automaton to the transition of a state to itself.

A pass is giving the ball to another player by throwing or kicking it. Note that in our two-

dimensional model of the field it is hard to distinguish the different modes of passing (throwing, kicking) the ball. To accomplish this, it would be necessary to somehow capture and analyse the body motion of the acting player, which is beyond the scope of this thesis. In case a player passes to the opponent team this is called a bad pass. The sequence of events is similar to the $Collect(p_1, p_2, p_3)$ predicate from before, with the difference, that the players don't have to stand next to each other, a pass is usually performed over a distance between two players. To allow both modes of passing the or ($'|'$) operator is introduced, which represents a branch on the timeline of the development. So either the predicate 'kick' or the predicate 'throw' can to be evaluated. The or ($'|'$) operator is associative and has a higher precedence than the $'\rightarrow'$ (then) operator.

$$Pass(p_1, p_2, b) \rightarrow Possess(p_1, b) \triangleright Kick(p_1, b) | Throw(p_1, b) \triangleright Catch(p_1, b) \triangleright Possess(p_2, b) \quad (5.5)$$

So three new sub predicates need to be defined:

- $Kick(p, b) \rightarrow meet(p, b) \vdash Disjoint(p, b)$
- $Throw(p, b) \rightarrow meet(p, b) \vdash Disjoint(p, b)$
- $Catch(p, b) \rightarrow Disjoint(p, b) \vdash meet(p, b)$

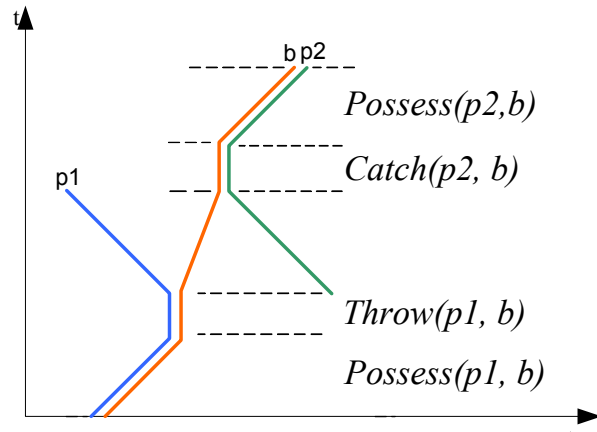


Figure 5.4.: Pass

The deduction from $\Pi = Pass(p_1, p_2, b)$ to a sequence of basic spatio-temporal predicates follows:

$$\begin{aligned}
C(\Pi) &= C(Possess \triangleright Kick | Throw \triangleright Catch \triangleright Possess) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash C(Kick | Throw \triangleright Catch \triangleright Possess) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash True \dashv (meet(p_1, b) \vdash Disjoint(p_1, b)) \mid \\
&\quad (meet(p_1, b) \dashv Disjoint(p_1, b)) \dashv C(Catch \triangleright Possess) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash True \dashv (meet(p_1, b) \vdash Disjoint(p_1, b)) \mid \\
&\quad (meet(p_1, b) \dashv Disjoint(p_1, b)) \dashv Disjoint(p_2, b) \vdash meet(p_2, b) \dashv C(Possess) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash True \dashv (meet(p_1, b) \vdash Disjoint(p_1, b)) \mid \\
&\quad (meet(p_1, b) \dashv Disjoint(p_1, b)) \dashv Disjoint(p_2, b) \vdash meet(p_2, b) \dashv Meet(p_2, b) \dashv \\
&\quad meet(p_2, b) \\
&\quad \exists(p_1, p_2) \in \mathbf{P}, p_1 \leq p_2, b \in B
\end{aligned}$$

(5.6)

qed.

Furthermore the activity of 'pass' is a repeated action. So the predicate contains a 'zero-to-many' repetition of passes, an operation covered by the star '*' operator:

$$Pass(p_1, p_2, b)^* \rightarrow Pass \triangleright Pass \triangleright \dots \quad (5.7)$$

Summary. In this section two new combinators were introduced, the repetition with the star '*' operator and the temporal branch with the *or* '|' operator. This notation is somehow similar to the EBNF notation. The predicate $Pass(p_1, p_2, b)$ has been decomposed in simpler activities and assembled as spatio-temporal development. Some simplifications have to be made.

5.2.3. Blocked

The activity "[role player] is blocked" is translated in this section. In a rugby game a player is blocked, when he can not move any further forward, 'cannot proceed' suggests that the player is in a movement. Remember the diagram of the 'kick-to-touch' game pattern, step 3, where the player moves forward, but is surrounded by opponents. Because no forward move is possible and tackle followed by a loss of ball possession is probable, the player kicks the ball out-of-bounds as soon as he gets aware of his unfortunate position. A player's notion of being in distress is very subjective. Different players will feel in different ways and stop their

run at different distances to the opponents.

This situation is difficult to recognize, because it involves to conclude from a topological state to a situation. Unlike the other predicates, this one captures a standstill situation as opposed to situations in motion.

To build a block it takes at least two players standing on the pitch and one player carrying the ball, running towards them and stopping his movement near them.

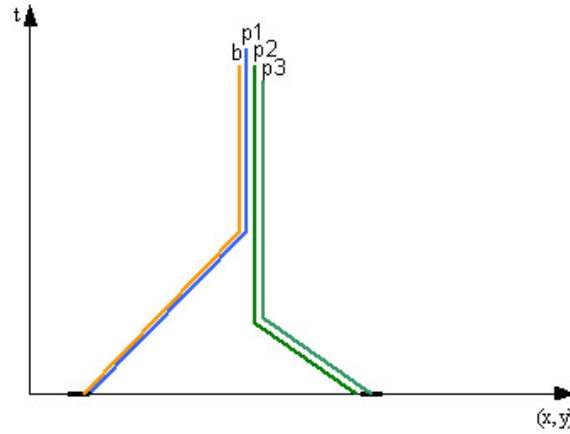


Figure 5.5.: *Blocked*

The space/time diagram sketch (Figure 5.5) shows two players (p_2, p_3) moving to a position and stopping there (building a block); the player in possession of the ball runs and stops in front of the block.

$$Blocked(p, b, t) \rightarrow Possess(p_1, b) \triangleright Runs(p_1) \triangleright Near(p_1, \exists p_i \in team_{\neg p_1}) \triangleright Stops(p_1) \quad (5.8)$$

At this point new types of predicates must be introduced. Till now, the used predicates rely on strict topological relationships. The new types extended the existing ones and also introduce a new concept. Firstly a threshold based predicate "Near" is defined, followed by "Stop" which works on the historical values of an object. At last a predicate called "Runs" represents a combination of both concepts.

The notion of being near somebody is very subjective. Here we define being near as sharing buffer space. To compute the predicate "Near" for the case where the time axis is frozen, the

distances between the object of interest and all others taken into account must be calculated and compared to the arbitrary threshold value.

The predicate "Stops" takes an object as argument and yields true as long as this object remains on the same position. To compute this at a time point t_i , we must know the location at t_{i-1} (which is the time point of the immediate previous tick). As long as the two positions are equal, the predicate evaluates to true. The paper in [3] refers to this kind of predicates - or queries in their terms - as IHC query (Intra History Cross Timestamp) or see Section 4.3 for more information.

Finally the predicate "Run" combines both concepts. To compute the actual speed, the current and the previous positions are needed. We insert the distance between the two points as s and the distance in time between t_0 and t_1 as time in Newton's equation of motion ($v = \frac{s}{t}$) and compare the resulting average velocity v to a threshold value for running (e.g. $v \leq 6km/h$ is walking, $v > 6km/h$ is running).

$$\begin{aligned}
C(\Pi) &= C(Possess(p_1, b) \triangleright Runs(p_1) \triangleright Near(p_1, \exists p_i \in team_{\neg p_1}) \triangleright Stops(p_1)) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash C(Runs(p_1) \triangleright Near(p_1, \exists p_i \in team_{\neg p_1}) \triangleright Stops(p_1)) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash Runs(p_1) \dashv C(Near(p_1, \exists p_i \in team_{\neg p_1}) \triangleright Stops(p_1)) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash Runs(p_1) \dashv true \vdash Near(p_1, \exists p_i \in team_{\neg p_1}) \dashv \\
&\quad C(Stops(p_1)) \\
&= Meet(p_1, b) \dashv meet(p_1, b) \vdash Runs(p_1) \dashv true \vdash Near(p_1, \exists p_i \in team_{\neg p_1}) \dashv \\
&\quad true \vdash Stops(p_1)
\end{aligned}$$

qed.

(5.9)

Summary. This section introduces threshold based predicates, historical predicates and a combination of them. Till now only basic spatio-temporal predicates based on the 9-intersection model [15] were discussed. These examples show how spatio-temporal predicates can be created to consider other interpretations than those based on topological relations. In this context predicate constriction and quantification are a big issue, for example no player stops for the whole duration of a game, but moves and stops. These predicate relations are constantly built and loosened.

5.2.4. KickOut

Next the activity "[role player] kicks [role ball] [role boundaries]" is taken into analysis. A player in possession of the ball kicks it at some stage and the ball leaves the pitch over a defined boundary. In the example's gameplay this is done after a blockage, so this is the starting predicate in the sequence.

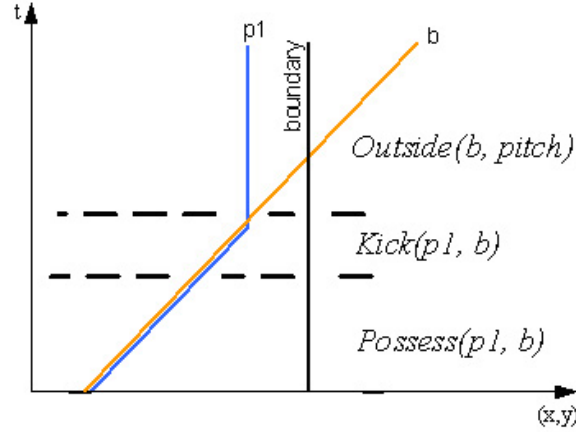


Figure 5.6.: KickOut

$$KickOut(p, b, r) \rightarrow Possess(p, b) \triangleright Kick(p, b) | Throw(p, b) \triangleright Outside(b, r) \quad (5.10)$$

$r \in boundaries$

Basically the predicate $Outside(b, pitch)$ computes a topological relation between a moving point and a fixed region. In the universe of a field sports game boundaries and special spots are not likely to change, on the contrary they have to be set according to proper rules. The main action of this activity is the ball's crossing of the outline, or in other words entering the region which is not the pitch (there are two regions: the pitch and everything around it). The entering and leaving (crossing) of a region is discussed in full detail in ([4], [7], [8]). The predicate $Outside(b, pitch)$ is thus similar to the predicate *Enter* in the literature.

$$\begin{aligned}
C(\Pi) &= C(Possess(p,b) \triangleright Kick(p,b) | Throw(p,b) \triangleright Outside(b,r)) \\
&= Meet(p,b) \dashv meet(p,b) \vdash C(Kick | Throw \triangleright Outside) \\
&= Meet(p,b) \dashv meet(p,b) \vdash True \dashv (meet(p,b) \vdash Disjoint(p,b)) | \\
&\quad (meet(p,b) \dashv Disjoint(p,b)) \dashv C(Outside) \\
&= Meet(p,b) \dashv meet(p,b) \vdash True \dashv (meet(p,b) \vdash Disjoint(p,b)) | \\
&\quad (meet(p,b) \vdash Disjoint(p,b)) \dashv true \vdash Disjoint(pitch,b) \dashv meet(pitch,b) \\
&\quad \vdash Inside(pitch,b) \\
&\quad p \in P, b \in B \\
&qed.
\end{aligned} \tag{5.11}$$

5.3. Assembling the pattern

Now that we have definitions for all four parts of the automaton in Figure 5.2, they are assembled to one pattern. Simple transitions are modelled by the 'then' compositor (see 4.2). The advanced combinator '**' is short for predicate repetition.

$$\begin{aligned}
KickToTouch(t_1, t_2, b, r) &\rightarrow Collect(p_9, p_i, b) \triangleright Pass(p_i, p_j, b)^* \triangleright \\
&\quad Blocked(p_j, b, t_2) \triangleright KickTo(p_j, b, outline) \\
&\quad b \in B; p_9, p_i, p_j \in t_1; i \neq j, i, j \in \{1 \dots 15\}
\end{aligned} \tag{5.12}$$

This is the kick-to-touch game pattern formulated as spatio-temporal development. By using the $C(\cdot)$ operator, one could deduct the sequence of basic spatio-temporal predicates forming the pattern.

5.4. Summary

This section develops step-by-step a specification of a rugby game pattern, formulated as a spatio-temporal development. The topological relationship between spatio-temporal objects can change like shown in the graph in Figure 4.3. The postulated specification is built by alternating sequences of spatio-temporal predicates. The basic spatio-temporal predicates as revised in Chapter 4 proved to be not sufficient, mainly because of their origin in the 9-intersection model and thus their focus on topological relations. New types of predicates were

demanded, which were introduced by a combination of evolution queries 4.3 and simple computations.

Table 5.2 lists all predicates used in this section. Column "Name" gives the predicate a speaking name, "Signature" notes the signature and under "Conditions" the domain sets are found. The "Level" at which a development resides is one higher than the highest of its composing predicates. This is a value denoting the complexity or the effort for a computation. The basic spatio-temporal predicates are on level 1.

Name	Level	Signature	Conditions
True	0	<i>True</i>	
False	0	<i>False</i>	
Meet	1	<i>Meet</i> (p, b)	$p \in \mathbf{P}, b \in \mathbf{B}$
Disjoint	1	<i>Disjoint</i> (p, b)	$p \in \mathbf{P}, b \in \mathbf{B}$
Outside	1	<i>Outside</i> (f, r)	$f \in \mathbf{F}, r \in \mathbf{D}$
Possess	2	<i>Possess</i> (p, b)	$p \in \mathbf{P}, b \in \mathbf{B}$
Grab	2	<i>Grab</i> (p_1, p_2, b)	$p_1, p_2 \in \mathbf{P}, p_1 \neq p_2$ $b \in \mathbf{B}$
Kick	2	<i>Kick</i> (p, b)	$p \in \mathbf{P}, b \in \mathbf{B}$
Throw	2	<i>Throw</i> (p, b)	$p \in \mathbf{P}, b \in \mathbf{B}$
Catch	2	<i>Catch</i> (p, b)	$p \in \mathbf{P}, b \in \mathbf{B}$
Pass	3	<i>Pass</i> (p_1, p_2, b)	$p_1, p_2 \in \mathbf{P}, p_1 \neq p_2$ $b \in \mathbf{B}$
Collect	3	<i>Collect</i> (p_1, p_2, b)	$p_1, p_2 \in \mathbf{P}, p_1 \neq p_2$ $b \in \mathbf{B}$
Runs	1	<i>Runs</i> (p)	$p \in \mathbf{P}$
Near	1	<i>Near</i> (p, t)	$\exists p \in \mathbf{T_R} \rightarrow t \in \mathbf{T_B} \vee$ $\exists p \in \mathbf{T_B} \rightarrow t \in \mathbf{T_R}$
Stops	1	<i>Stops</i> (p)	$p \in \mathbf{P}$
Blocked	2	<i>Blocked</i> (p, b, t)	$\exists p \in \mathbf{T_R} \rightarrow t \in \mathbf{T_B} \vee$ $\exists p \in \mathbf{T_B} \rightarrow t \in \mathbf{T_R}$

B...ball **D**...boundaries **F**...figures
P...players **T_B**...team blue **T_R**...team red

Table 5.2.: Overview of Kick-to-touch predicates

6. Implementing the concepts

Here the concepts from the last sections are put together. A sample realisation in Prolog gives an outline how to tackle the task of implementing the predicates and performing their evaluation. The whole procedure of computing a spatio-temporal predicate is explained from the start where a user gives an input till the end where the result of a computed function exists in the memory.

The first section discusses the steps to be taken to perform the computation of a spatio-temporal predicate.

6.1. Phases of specification and computation

The computation a spatio-temporal development is broken down to three consecutive phases:

1. *Formalization of specification:* Describe the real-world events with a formalism. Construct an abstract model.
2. *Compilation of the model:* Build a computable function from the abstract model by finding an interpretation.
3. *Execution:* Matching against a dataset. Apply the function to an input dataset.

A possible forth phase is the collection of data, which is skipped, because it has no direct impact on the implementation. This phase can be performed before or in parallel to the three phases of the computation (retrospective vs. simultaneous).

6.1.1. Formalization of specification

At this stage the real-world events are formalized. This has to be done by a human user, because this task is hard to automate. The user has to have an understanding of the game he wants to analyse and he has to know how the predicates work. Furthermore he must know the

goal of the query, which kind of answer he wants and what he can get from the system.

To support the user to input his specification several possibilities exist. There are formal query languages with some spatial-temporal capabilities like in [1] or [9]. To stress the importance of spatio-temporal predicates in this context, it is imaginable to formulate a proper predicate language. A very promising approach is done in [7], a paper, which presents some fundamental ideas how to use a visual query language as a frontend to build spatio-temporal developments.

Summary. At the end of this phase a set of combined and syntactically correct predicates are present.

6.1.2. Compilation of the model

The task of this phase is to build a computable function from the abstract model. The predicates defined during the formalization phase are broken down to the basic and atomic predicates and assigned some semantics. In Figure 6.1 the breakdown of the "kick to touch"-development from Chapter 5 is shown. At this stage an optimization of the development is possible. For example, similar predicates at the beginning and end of neighbouring developments can be eliminated, or data structures can be shared.

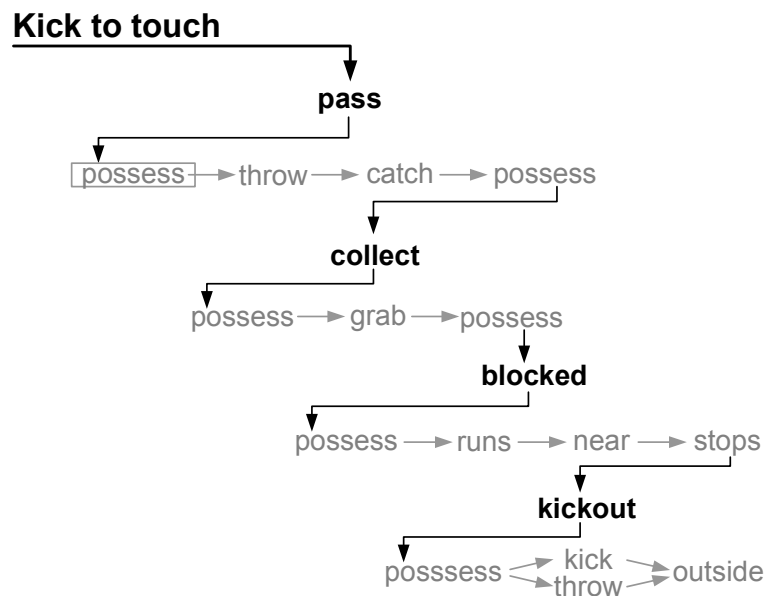


Figure 6.1.: *Kick-to-touch predicate order*

The result of this phase is a computable function. This function follows some interface specification, which specify properties of the input data and other necessities for a computation. In this context a computable function defines not only *what* to compute, but also *how* to compute things. In this sense such a function incorporates algorithms and data structures. In the end a function in the form of machine executable code is generated.

Summary. This phase does an optimization and assigns evaluation algorithms to predicate definitions.

6.1.3. Matching against a dataset

At last the compiled function mapping a spatio-temporal predicate is executed with an input database. There are two requirements for function and data: the data is available in a predefined form, for example in a database following some schema and the function is aware of the schema and can work on it. If both requirements are met, the function can be forwarded to a processor for execution and the result can be stored in the memory. In the context of this thesis, this procedure is called the "matching of a predicate against a dataset".

Summary. The last phase applies the function to an input dataset and stores the result.

6.2. Theoretic concept to implementation in Prolog

After the first phase is finished, the user decides on which predicates to use for his query. They are passed on to the system in phase two, in which they are broken down to atomic spatio-temporal predicates and submitted to the next phase. In phase three these predicates are actually computed and reassembled to build the result of the operation. These two stages of computing and reassembling are reflected in the implementation.

6.2.1. The data

The data is organised as Prolog facts. A simplification of the Tripod data model's history serves as data structure: $H = \{\mathbb{N}^2, \text{TimeIntervals}, 1, \{\langle \tau, \sigma \rangle\}_{i \in \mathbb{I} \subset \mathbb{N}}\}$. It reads: i pairs consisting of coordinates (\mathbb{N}^2) and time intervals of length one form a history. This history is implemented as a list holding the object's positions at timesteps of length one:

```

playerpos(player1, [position(0,0), position(1,1), position(2,2),
                    position(2,2), position(1,3), position(0,4),
                    position(0,4), position(0,4)]).

```

To facilitate the implementation, an explicit `TimeIntervals` value is replaced through the implicit position in the list. Thus the first element in the list `position(0,0)` is valid during the time interval $[0,1)$, the second element `position(1,1)` during $[1,0)$, ...

6.2.2. First stage: computation

In 4.7 a quick look at a predicate's mode of operation was given. Here this idea is pursued in more detail. A predicate is implemented as a Prolog goal and takes some objects as arguments. It queries the database for the object's positions and computes the predicate for each instant. The result of a spatio-temporal predicate is a temporal Boolean $\langle e_0, e_1, \dots, e_n \rangle$, $e_i \in \{T, F\}, 0 \leq i \leq n$, which is a vector of truth values. Each element e_i of the vector reflects the result of the predicate at instant i .

As an example the implementation of predicate '*Meet*' is presented. First the database - in this example the prolog facts - is queried for the needed histories (PL1 and PL2). In the next step either goal `meet3` or `meet4` is called depending on the number of arguments. This recursive goal goes to the end of the list and then builds the result vector. If the two structures A and B are the same ($A==B$), the result of the predicate for this time-slice is `true`. Otherwise if A and B are different ($A \neq B$), the value appended to the result vector is `false`.

```

meet3([], [], []).
...
meet3([A|P], [B|Q], L) :- meet3(P, Q, NL), A==B, append([true], NL, L).
meet3([A|P], [B|Q], L) :- meet3(P, Q, NL), A \== B, append([false], NL, L).
...
meet(P1, P2, L) :- playerpos(P1, PL1), playerpos(P2, PL2), meet3(PL1, PL2, L).
meet(P1, P2, B, L) :- playerpos(P1, PL1), playerpos(P2, PL2), playerpos(B, BL),
    meet4(PL1, PL2, BL, L).

```

So the result of the computation stage is a vector of truth values, for example the evaluation of predicate `meet(p1, b)` yields:

```
meet(pl,b) = < true,true,true,true,true,false,false >
```

Several of these temporal Booleans are used as input for the next stage.

6.2.3. Second stage: reassembly

During the second stage, the temporal Booleans are reassembled to validate the predicate. The main data structure used here is a matrix of string values, where each boolean vector is a row. The rows are ordered by the occurrence of the corresponding predicate in the development. Several predicates are chained together by 'then' compositors. The order in the chain determines the order of the rows: the earlier predicates stand on top, the later ones on the bottom. The columns of this matrix shows then, at which point in time the composing predicates are true or false.

At this point the problem of evaluating a consecutive set of spatio-temporal predicates is transformed into a string search problem: The evaluation algorithm hops from one true value to the other and searches a path from the left upper corner to the right bottom corner in the matrix. If such a path exists, the evaluation algorithm's prolog goal is successful. Thus the spatio-temporal predicate/development itself is successful. As a consequence of time being linear, only hops from the left to the right are valid and - since the predicates are executed in a consecutive order - from the top downwards.

The Prolog code effects the evaluation of a chain of spatio-temporal predicates and reads as follows:

```
walk(DL,C,R) :- getrc(DL,R,M),C>=M, write('success\r\n').
walk(DL,C,R) :- NC is C+1, NR is R+1, getval(DL,NR,NC,A), A==true,
                jump(NR,NC), walk(DL,NC,NR).
walk(DL,C,R) :- NC is C+1, getval(DL,R,NC,A), A==true, jump(R,NC),
                walk(DL,NC,R).
```

If the rightmost column is exceeded and the last row is reached, the algorithm terminates successfully. Otherwise the algorithm looks for the next matrix value to visit. First it tries to hop to the next predicate, which is one to the right for the next instant and one down for the next predicate. If this is not successful the next instant (one right) in the row is checked and - if true - visited. So if the algorithm cannot proceed with the next predicate, it tries to proceed

with the current one. If either of them fail, there is a gap in the succession of the predicate. and thus the algorithm fails. Of course more than one path through the matrix can exist.

Figures 6.2, 6.3, 6.4 and 6.5 show some examples from the last section. The temporal Booleans of the predicates are aligned and the emphasised values represent the field of operation for the algorithm. Possible paths are marked by pulled through arrows. In Figure 6.4 two paths are marked. during this development two predicates are valid in parallel. So there is a one-elemental chain and one three-elemental chain to be resolved by the algorithm.

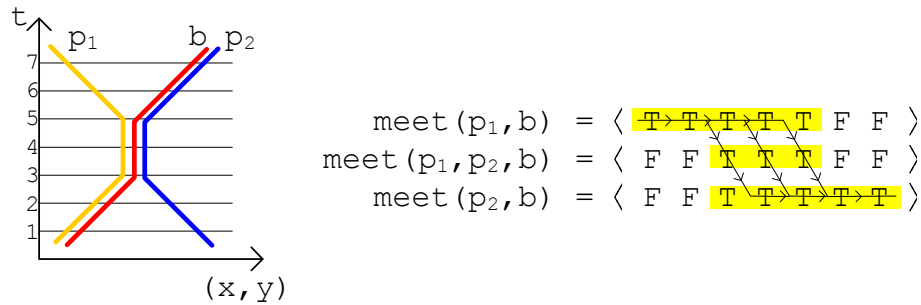


Figure 6.2.: *Temporal Boolean for predicate 'collect'*

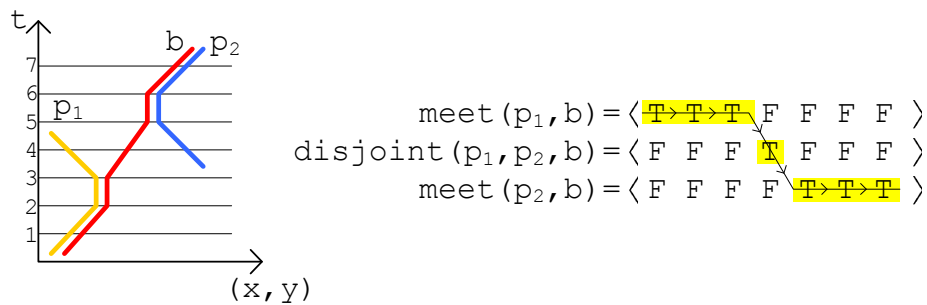


Figure 6.3.: *Temporal Boolean for predicate 'pass'*

6.3. Executing the prolog code

The resulting temporal Booleans of the predicates computations are taken as rows of a matrix like explained above. Then all possible paths from the upper left corner of the matrix to the lower right are found. Reading the matrix in this way corresponds to an execution of the predicates in temporal succession. The algorithm first goes into depth and then finds the other

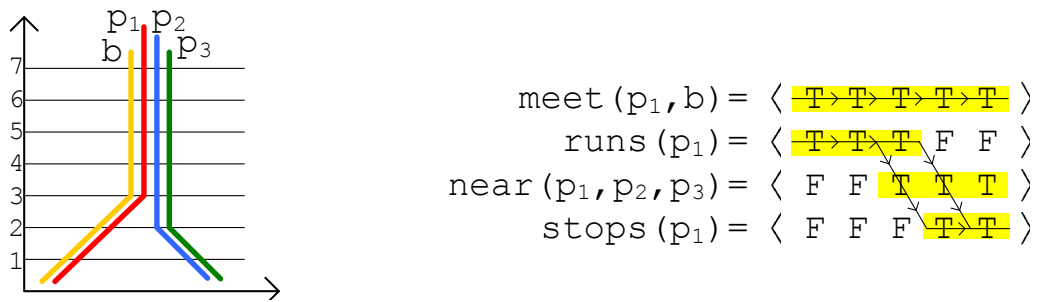


Figure 6.4.: Temporal Boolean for predicate 'blocked'

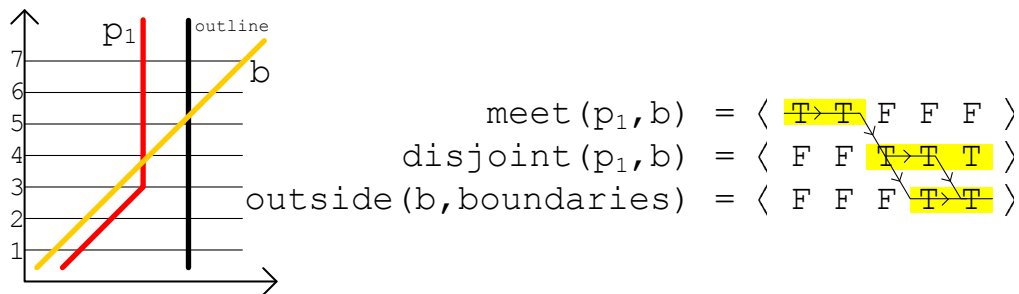


Figure 6.5.: Temporal Boolean for predicate 'kickout'

solutions to the graph search problem when backtracking.

In the output of the goal `rX` stands for *row number X* and `cY` is for *column number Y*. After the first solution is found, the prolog system probes for the next while backtracking. If there are other solutions possible, after one has been found, the system prompts `true ?`. The user can either tell the system to show more solutions (enter `;`) or stop the execution (enter `<return>`).

6.3.1. Calling goal: collect

In this part of the example it is easy to notice, that the graph search problem has multiple solutions. For the considered spatio-temporal problem, only one solution is needed to show, that the whole predicate is successful. The spatio-temporal predicate `collect` has six possible solutions. When probing for a seventh, the system fails and terminates the execution.

The input data is encoded in Prolog facts:

```
playerpos(player3, [position(0,0), position(1,1),
```

```
position(2,2), position(2,2), position(2,2),
position(1,3),position(0,4)]).
```

```
playerpos(player4, [position(4,0), position(3,1),
position(2,2), position(2,2), position(2,2),
position(3,3),position(4,4)]).
```

```
playerpos(ball2, [position(0,0), position(1,1),
position(2,2), position(2,2), position(2,2),
position(3,3),position(4,4)]).
```

And one execution of the goal predicate yields this output:

```
GNU Prolog 1.2.18
```

```
By Daniel Diaz
```

```
Copyright (C) 1999-2004 Daniel Diaz
```

```
| ?- test_collect.
```

```
Calling goal: collect(player3,player4,ball2).
```

```
r1 c1  r1 c2  r2 c3  r3 c4  r3 c5  r3 c6  r3 c7  success
```

```
true ? ;
```

```
r2 c4  r3 c5  r3 c6  r3 c7  success
```

```
true ? ;
```

```
r2 c5  r3 c6  r3 c7  success
```

```
true ? ;
```

```
r1 c3  r2 c4  r3 c5  r3 c6  r3 c7  success
```

```
true ? ;
```

```
r2 c5  r3 c6  r3 c7  success
```

```
true ? ;
```

```
r1 c4  r2 c5  r3 c6  r3 c7  success
```

```
true ? ;
```

```
r1 c5
```

```
(3 ms) no
```

```
| ?-
```

6.3.2. Calling goal: pass

The execution of this predicates is straight forward. Only one solution is possible and only one path is found.

The input data is encoded in Prolog facts:

```
playerpos(player1, [position(0,0), position(1,1),
                    position(2,2), position(2,2), position(1,3),
                    position(0,4),position(0,4),position(0,4)]).
```

```
playerpos(player2, [position(8,0), position(8,0),
                    position(7,1), position(6,2), position(5,3),
                    position(4,4),position(4,4),position(5,5)]).
```

```
playerpos(ball, [position(0,0), position(1,1),
                 position(2,2), position(2,2), position(3,3),
                 position(4,4),position(4,4),position(5,5)]).
```

And one execution of the goal predicate yields this output:

```
GNU Prolog 1.2.18
```

```
By Daniel Diaz
```

```
Copyright (C) 1999-2004 Daniel Diaz
```

```
| ?- test_pass.
```

```
Calling goal: pass(player1,player2,ball).
```

```
r1 c1  r1 c2  r1 c3  r1 c4  r2 c5  r3 c6  r3 c7  r3 c8  success
```

```
true ? ;
```

```
(1 ms) no
```

```
| ?-
```

6.3.3. Calling goal: blocked

Here the algorithm has to be called twice, since there are two actions in parallel: The player possesses the ball and runs and stops. These two parts are evaluated separately and the respective results are tied with a logical and operation. This is expressed in Prolog through tying the right hand goals with colons. Writing several clauses for one goal is then read as logical or.

The input data is encoded in Prolog facts:

```
playerpos(player7, [position(0,0), position(3,3),
    position(6,6), position(9,9), position(9,9)]).
```

```
playerpos(player8, [position(18,18), position(15,15),
    position(12,12), position(10,10), position(10,10)]).
```

```
playerpos(player9, [position(18,17), position(15,14),
    position(12,11), position(10,9), position(10,9)]).
```

```
playerpos(ball3, [position(0,0), position(3,3), position(6,6),
    position(9,9), position(9,9)]).
```

And one execution of the goal predicate yields this output:

```
GNU Prolog 1.2.18
By Daniel Diaz
Copyright (C) 1999-2004 Daniel Diaz
| ?- test_blocked.
Calling goal: blocked(player7,player8,player9,ball3).
r1 c1  r1 c2  r1 c3  r1 c4  r1 c5  success
r2 c1  r2 c2  r2 c3  r3 c4  r4 c5  success

true ? ;
r3 c5  success

true ? ;

(2 ms) no
```

| ?-

6.4. Summary and outlook

This section shows, how the concept of a spatio-temporal predicate can be implemented using a Prolog system. Two main phases of computation and reassembly were identified and a sample implementation was done, in order to show how the predicates are put into work. After finding a solution for the spatial relations, the validation of the temporal succession was transformed into a string search problem and the solution was then transferred back. Starting with some facts representing the input position data of players and a ball, the predicates were deducted. If a predicate's prolog goal is successful, the predicate is satisfied and if the prolog goal fails the predicate fails.

This way of deducting predicates and thus obtaining a boolean as a result is not the only way of computing spatio-temporal relations. Another possibility is the use of functions rather than predicates. Such a function takes a database as input and yields a subset of this database as a result, thus it acts as filter. This subset can reveal more detailed information on the situation like for example the length of a player's route *during* a situation. Spatio-temporal filters cut out portions of a spatio-temporal dataset, and combined with some evaluation functions, they represent a powerful way of penetrating spatio-temporal data. Since predicates are functions, which have the boolean set as image set ($f : A \rightarrow \mathbb{B}$), they are a special case of a filter. If they are satisfied, then a subset containing the specified situation exists in the database. In practice, the computation of a spatio-temporal filter is similar to the evaluation of a spatio-temporal predicate: the portion of the dataset, where the predicate is fulfilled, is cut out for all participating objects in the specified situation. Thus the aggregation is replaced through a copy operation.

7. Conclusion

This work discussed the application of spatio-temporal predicates to effect situation recognition in field sports. It shows how the framework of spatio-temporal predicates ([6]) can be adapted to create domain-specific predicates. Basic predicates and combinators from the canonical set are taken and given some semantics from the domain of field sports. This way an application-dependent subset of predicates is defined. A link is made to the topic of spatio-temporal patterns [4].

An example game pattern is explained and stepwise broken down to its rudimentary actions. These are then translated into the language of spatio-temporal predicates. The single predicates are wired by sequential or parallel combinators to form a development. This development serves as specification of a situation, which can be matched against an appropriate database. A development hides the underlying predicates and is a predicate itself. The use of speaking names for predicates facilitates an intuitive way for specifying situations. Automata are suggested as possible visualisation and to facilitate the intuitive design of developments.

A basic implementation was made and the main steps to build a system for spatio-temporal predicates are discussed. The spatio-temporal predicates were mapped to Prolog goals and the validation was executed as a deduction from facts representing spatio-temporal data. For a realization there are many challenges to be met and problems to be solved. A big topic is to find proper user-interfaces to specify developments. In Chapter 5 tactical drawings of a sports team is taken as a spatio-temporal map. A visual query language derived from such a drawing is an intuitive way of specifying queries and one which many users have already executed.

This work shows how the adaption of spatio-temporal predicates is used to penetrate a spatio-temporal dataset in order to reveal inherent information. The topic of field sports has been chosen, because the special properties of sport make it especially practical to study spatio-temporal relations. A field sports game can be understood as a system and spatio-temporal predicates represent one way to give a formal description of such a system.

A. Acronyms

ATP	Association of Tennis Professionals
Cairos	
CHC	Cross-History Cross-timestamp
CHI	Cross-History Intra-timestamp
DBMS	Database Management System
DOMINO	Databases fOr MovINg Objects tracking [18]
EBNF	Extended Backus Naur Form
IHC	Intra-History Cross-timestamp
LPM	Local Position Measurement
ODMG	Object Database Management Group
ROSE	RObust Spatial Extension
SDT	Spatial Data Type
SQL	Structured Query Language

B. Bibliography

- [1] Michael Böhlen, Christian S. Jensen, and Bjorn Skjellaug. Spatio-temporal database support for legacy applications. *Proceedings of the 1998 ACM Symposium on Applied Computing, Atlanta, Georgia*, pages 226–234, 1998.
- [2] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
- [3] Nassima Djafri, Alvaro A. A. Fernandes, Norman W. Paton, and Tony Griffiths. Spatio-temporal evolution: querying patterns of change in databases. *Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 35–41, 2002.
- [4] Martin Erwig. Toward spatiotemporal patterns. In De Caluwe et al., editor, *Flexible Querying and Reasoning in Spatio-Temporal Databases: Theories and Applications*, pages 29–54. Springer Verlag, 2004.
- [5] Martin Erwig, Ralf Hartmut Güting, Markus Schneider, and Michalis Varzirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.
- [6] Martin Erwig and Markus Schneider. Spatio-temporal predicates. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):881–901, July 2002.
- [7] Martin Erwig and Markus Schneider. A visual language for the evolution of spatial relationships and its translation into a spatio-temporal calculus. *Journal of Visual Languages and Computing*, 14(2):181–211, 2003.
- [8] Güting et al. Spatio-temporal models: An approach based on data types. In T. Sellis et al., editor, *Spatio-Temporal Databases - The Chorochronous Approach. LNCS 2520*, pages 117–176. Springer Verlag, 2003.

- [9] Tony Griffiths, Alvaro A. A. Fernandes, Norman W. Paton, and Robert Barr. The tripod spatio-historical data model. *Data and Knowledge Engineering*, 49(1):23–65, 2004.
- [10] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 4(1):1–42, 2000.
- [11] Ralf Hartmut Güting and Markus Schneider. Realm-based spatial data types: Rose algebra. *VLDB Journal*, 4:100–143, 1995.
- [12] Torsten Hägerstrand. What about people in spatial science? *Papers of the Regional Science Association (RSAI)*, 24:7–21, 1970.
- [13] Torsten Hägerstrand. Space, time and human conditions. In Anders Karlqvist, L Lundqvist, and Folke Snickars, editors, *Dynamic allocation of urban space*, pages 3–12. Farnborough: Saxon House, 1975.
- [14] J. Loeckx, H. D. Ehrich, and M. Wolf. *Specification of abstract data types*. John Wiley and sons, Inc. and B.G. Teubner Publishers, 1996.
- [15] Egenhofer MJ and Herring JR. Categorizing binary topological relationships between regions, lines and points in geographic database. Technical Report 91-7, University of Maine, 1991.
- [16] Antoni Moore, Peter Whigham, Colin Aldridge, Alec Holt, and Ken Hodge. Spatio-temporal and object visualization in rugby union. *The Information Science Discussion Paper Series*, (2002/03), June 2002.
- [17] Antoni Moore, Peter Whigham, Alec Holt, Colin Aldridge, and Ken Hodge. A time geography approach to the visualisation of sport. *Proceedings of the 7th International Conference on GeoComputation, University of Southampton*, 2003.
- [18] Ouri Wolfson, A. Prasad Sistla, Bo Xu, Jutai Zhou, and Sam Chamberlain. Domino: Databases for moving objects tracking. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 547–549. ACM Press, 1999.
- [19] Adnan Yazici, Öznur Yavuz, and Roy George. An mpeg-7 based video database management system. In De Caluwe et al., editor, *Flexible Querying and Reasoning in Spatio-Temporal Databases: Theories and Applications*, pages 181–210. Springer Verlag, 2004.

C. Sample implementation in Prolog

```

%=====
% THE DATA REPRESENTED AS FACTS
%=====

%-----FORMAT-----

make_position(X,Y,position(X,Y)).
get_x(position(X,_),X).
get_y(position(_,Y),Y).

%-----PASS-----

playerpos(player1, [position(0,0), position(1,1), position(2,2), position(2,2),
                    position(1,3),position(0,4),position(0,4),position(0,4)]).

playerpos(player2, [position(8,0), position(8,0), position(7,1), position(6,2),
                    position(5,3),position(4,4),position(4,4),position(5,5)]).

playerpos(ball, [position(0,0), position(1,1), position(2,2), position(2,2),
                 position(3,3),position(4,4),position(4,4),position(5,5)]).

%-----COLLECT-----

playerpos(player3, [position(0,0), position(1,1), position(2,2), position(2,2),
                    position(2,2),position(1,3),position(0,4)]).

playerpos(player4, [position(4,0), position(3,1), position(2,2), position(2,2),
                    position(2,2),position(3,3),position(4,4)]).

playerpos(ball2, [position(0,0), position(1,1), position(2,2), position(2,2),
                  position(2,2),position(3,3),position(4,4)]).

%-----BLOCKED-----

playerpos(player7, [position(0,0), position(3,3), position(6,6), position(9,9),
                    position(9,9)]).

playerpos(player8, [position(18,18), position(15,15), position(12,12),
                    position(10,10), position(10,10)]).

playerpos(player9, [position(18,17), position(15,14), position(12,11),
                    position(10,9),position(10,9)]).

playerpos(ball3, [position(0,0),position(3,3),position(6,6),position(9,9),
                  position(9,9)]).

%=====
% SOME MORE OR LESS USEFUL TOOLS
%=====

%-----TOOLS-----

revert([], []).
revert([H|T],S) :- revert(T,NS), append([H],NS,S).

invert([], []).
invert([H|T],S) :- invert(T,NS), H==true, append([false],NS,S).
invert([H|T],S) :- invert(T,NS), H==false, append([true],NS,S).

same(A,B,_) :- A==B.
same(A,_,C) :- A==C.

```

```

same(_,B,C) :- B==C.

op_or(A,B,X) :- A==true, B==true, X=true.
op_or(A,B,X) :- A==true, B==false, X=true.
op_or(A,B,X) :- A==false, B==true, X=true.
op_or(A,B,X) :- A==false, B==false, X=false.

op_and(A,B,X) :- A==true, B==true, X=true.
op_and(A,B,X) :- A==true, B==false, X=false.
op_and(A,B,X) :- A==false, B==true, X=false.
op_and(A,B,X) :- A==false, B==false, X=false.

myfirst(F, [F|_]).
myrest(R, [_|R]).

fold_or([],false).
fold_or([A|T],V) :- fold_or(T,NV), op_or(A,NV,V).

fold_and([],true).
fold_and([A|T],V) :- fold_and(T,NV), op_and(A,NV,V).

merge([],[],[]).
merge([H1|T1],[H2|T2],L) :- merge(T1,T2,NL), op_and(H1,H2,A), append([A],NL,L).

not(P) :- call(P), !, fail.
not(_).

equal3(A,B,C) :- A==B, B==C, A==C.
nequal3(A,B,C) :- A\==B, B\==C, A\==C.

eqlist(A,B,C,L) :- equal3(A,B,C), append([true],[],L).
eqlist(A,B,C,L) :- not(equal3(A,B,C)), append([false],[],L).
neqlist(A,B,C,L) :- not(equal3(A,B,C)), append([true],[],L).
neqlist(A,B,C,L) :- equal3(A,B,C), append([false],[],L).

doublend(A, [], [A]).
doublend(_, [H|T], [H|L]) :- doublend(H,T,L).

distance(P,Q,D) :- get_x(P,Px), get_y(P,Py), get_x(Q,Qx), get_y(Q,Qy),
    D is sqrt((Px-Qx)*(Px-Qx)+(Py-Qy)*(Py-Qy)).

velocity(S,T,V) :- V is S/T.
velo_list([],[]).
velo_list([_],[]).
velo_list([F|T],L) :- velo_list(T,NL), myfirst(S,T), distance(F,S,D),
    velocity(D,1,N), append([N],NL,L).

%=====
% EVALUATING 1: Build the strings
%=====
%-----MEET-----

meet3([],[],[]).
meet3([_],[],[]).
meet3([],[_],[]).
meet3([A|P],[B|Q],L) :- meet3(P,Q,NL), A==B, append([true],NL,L).
meet3([A|P],[B|Q],L) :- meet3(P,Q,NL), A\==B, append([false],NL,L).

meet4([],[],[],[]).
meet4([A|P],[B|Q],[C|R],L) :- meet4(P,Q,R,NL),
    equal3(A,B,C),

```

```

        append([true],NL,L).

meet4([A|P],[B|Q],[C|R],L) :- meet4(P,Q,R,NL),
                               not(equal3(A,B,C)),
                               append([false],NL,L).

meet(P1,P2,L) :- playerpos(P1,PL1), playerpos(P2,PL2), meet3(PL1,PL2,L).
meet(P1,P2,B,L) :- playerpos(P1,PL1), playerpos(P2,PL2), playerpos(B,BL),
                  meet4(PL1,PL2,BL,L).

testmeet(L) :- playerpos(player1,PL1), playerpos(player2,PL2), playerpos(ball,
PL3),
              meet4(PL1,PL2,PL3,L).

%-----DISJOINT-----

disjoint3([],[],[]).
disjoint3([_],[],[]).
disjoint3([],[_],[]).

disjoint3([A|P],[B|Q],L) :- disjoint3(P,Q,NL), A==B, append([false],NL,L).
disjoint3([A|P],[B|Q],L) :- disjoint3(P,Q,NL), A\==B, append([true],NL,L).

disjoint4([],[],[],[]).
disjoint4([A|P],[B|Q],[C|R],L) :- disjoint4(P,Q,R,NL),
                                   nequal3(A,B,C),
                                   append([true],NL,L).

disjoint4([A|P],[B|Q],[C|R],L) :- disjoint4(P,Q,R,NL),
                                   not(nequal3(A,B,C)),
                                   append([false],NL,L).

disjoint(P1,P2,L) :- playerpos(P1,PL1), playerpos(P2,PL2), disjoint3(PL1,PL2,L).
disjoint(P1,P2,B,L) :- playerpos(P1,PL1), playerpos(P2,PL2), playerpos(B,BL),
                      disjoint4(PL1,PL2,BL,L).

testdis(L) :- playerpos(player1,PL1), playerpos(player2,PL2), playerpos(ball,
PL3),
              disjoint(PL1,PL2,PL3,L).

%-----STOPS-----

stops1([],[]).
stops1([H|T],L) :- stops1(T,NL), H=0.0, append([true],NL,L).
stops1([H|T],L) :- stops1(T,NL), H\=0.0, append([false],NL,L).

stops(P,L) :- playerpos(P,PL), velo_list(PL,VL), stops1(VL,TL), doublend(_,TL,L).

%-----NEAR-----

near3(A,B,C) :- distance(A,B,X),X<1.5, distance(A,C,Y),Y<1.5,
                  distance(B,C,Z),Z<1.5.

near_list([],[],[],[]).
near_list([A|P],[B|Q],[C|R],L) :- near_list(P,Q,R,NL), near3(A,B,C),
                                   append([true],NL,L).
near_list([A|P],[B|Q],[C|R],L) :- near_list(P,Q,R,NL), not(near3(A,B,C)),
                                   append([false],NL,L).

near(A,B,C,L) :- playerpos(A,L1), playerpos(B,L2), playerpos(C,L3),
                  near_list(L1,L2,L3,L).

```

```

%-----RUNS--

runsl([],[]).
runsl([H|T],L) :- runsl(T,NL), H>=4.0, append([true],NL,L).
runsl([H|T],L) :- runsl(T,NL), H>=0.0, H<4.0, append([false],NL,L).

runs(P,L) :- playerpos(P,PL),velo_list(PL,VL),runsl(VL,TL),doublend(_,TL,L).

%=====
% EVALUATING 2: Search the strings
%=====

%-----GETVAL--

count([],0).
count([_|T],S) :- count(T,NS), S is NS+1.

nth_member(1,[M|_],M).
nth_member(N,[_|T],M) :- N>1, N1 is N-1, nth_member(N1,T,M).

getval(DL,R,C,V) :- nth_member(R,DL,L), nth_member(C,L,V).
getrc(DL,R,V) :- nth_member(R,DL,L),count(L,V).

jump(R,C) :- write('r'),write(R),tab(1),write('c'),write(C),tab(2).

%-----WALK--

walk(DL,C,R) :- getrc(DL,R,M),C>=M, write('success\r\n').
walk(DL,C,R) :- NC is C+1, NR is R+1, getval(DL,NR,NC,A), A==true,
jump(NR,NC), walk(DL,NC,NR).
walk(DL,C,R) :- NC is C+1, getval(DL,R,NC,A), A==true, jump(R,NC),
walk(DL,NC,R).

%=====
% THE PREDICATES
%=====

%-----PASS--

pass(P1,P2,B) :- meet(P1,B,L1), disjoint(P1,P2,B,L2), meet(P2,B,L3), jump(1,1),
walk([L1,L2,L3],1,1).

%-----COLLECT--

collect(P1,P2,B) :- meet(P1,B,L1), meet(P1,P2,B,L2), meet(P2,B,L3), jump(1,1),
walk([L1,L2,L3],1,1).

%-----BLOCKED--

blocked(P1,P2,P3,B) :- meet(P1,B,L1), jump(1,1), walk([L1],1,1),
runs(P1,L2), near(P1,P2,P3,L3), stops(P1,L4), jump(2,1),
walk([L1,L2,L3,L4],1,2).

```

```
%=====
% EXECUTION TESTS
%=====

%-----EXECUTION-----

test_pass :- write('Calling goal: pass(player1,player2,ball).\r\n'),
             pass(player1,player2,ball).

test_collect :- write('Calling goal: collect(player3,player4,ball2).\r\n'),
               collect(player3,player4,ball2).

test_blocked :- write('Calling goal: blocked(player7,player8,player9,
ball3).\r\n'),
               blocked(player7,player8,player9,ball3).

%-----
```