Chapter 5

Design of a Qualitative Description Language

5.1 Introduction

A qualitative description language has been designed for representation of motion patterns. This language allows us to describe motion patterns that we want to confirm / recognize in an input stream. Concepts like *binary motion pattern*, *basic multi object pattern* and *multi object pattern* are treated as types in this language. A criticism of learning using formal grammar is the tediousness of the process. QDL can alleviate this problem by providing an easy way of pattern specification. A feature called *skip transition*, included in this language, allows us make a representation coarser. Another drawback of grammar based learning is the representational difficulty of concurrency among patterns. QDL can represent concurrency among motion patterns using Allen's interval algebra [18].

A description written in QDL is syntactically analysed and an intermediate representation is generated. This intermediate representation is interpreted over input for recognition of patterns. An object is represented as a variable inside a QDL description and data for every variable are read from input stream. Then, recognition of a binary motion pattern becomes equivalent to type checking.

5.2 Significance

In chapter 4, we have discussed how the framework can learn definitions of binary and basic multi object movement patterns from training data. In Figure 4.5, we have shown how a basic multiobject object pattern is structured. Binary qualitative relations (abbreviated as *bqr* in the figure) for different movement parameters are computed and primitive patterns are constructed out of these qualitative relations. A binary motion pattern consists of a number of primitives. Finally, a basic multiobject object pattern is made out of one or more binary motion patterns. Algorithms for learning as well as recognition of binary and basic multiobject object patterns were presented.

The qualitative description language (**QDL**), proposed in this thesis, addresses the issue of hierarchical construction of motion patterns. Hierarchical representation of motion pattern is related with creation of conceptual groupings of objects in the pattern. Each such grouping is an encapsulation of a set of objects and their temporal relationships. Temporal relations between these groupings is another important aspect of a hierarchy. A basic multiobject object pattern is one such grouping that is automatically learnable from input. QDL treats **BMOP**s as the basic unit of creating a hierarchical representation. This means that any hierarchically constructed motion pattern should be ultimately decomposible to a set of basic multiobject object patterns. It is so because **BMOP**s are the units that are learnable from input.

The language has another important use. Sometimes, we may not be interested to know whether the entire pattern has occurred. Instead, we may be interested in recognition of a particular pattern among certain objects. For example, in a motion pattern among multiple persons, we may like to know the pattern between two particular persons. In this context, the qualitative language can play the role of a query language.

5.3 Language Design

5.3.1 Basic Concepts

A program, written in **QDL**, specifies the definition of a motion pattern. Use of description languages is reported in other domains to represent the structure of various entities. Mention may be made of hardware description languages and **XML**. In these languages, the structure of input is parsed according to a program specification. **QDL** is different from these languages in certain aspects. This difference comes from characteristics of the motion pattern recognition problem. Motion pattern recognition too requires structure recognition, but there is a temporal dimension to be considered. It is not enough to identify the structure alone. The temporal relationship between elements of the structure is an equally important necessity. Secondly, motion pattern recognition, proposed in this thesis, requires analysis of structure as well as analysis of input data values. For example, for handling error, we need to know the exact primitives that have occurred in succession i.e. we need to analyse the data values that have occurred in input.

QDL uses type theory of programming languages. Motion patterns, binary as well as multiobject object, are treated as types. Variables can be declared to be of a *motion pattern type*. Type declaration is done at compile time. Each variable is associated with an object. At the time of execution, values for these variables are read from input data stream. If a variable gets a value of correct type, then the corresponding motion pattern is recognised. Otherwise, there is a type mismatch at run time and the motion pattern is not recognised. Temporal constraints between motion patterns are expressed using operators based on Allen's interval algebra. For recognition of a pattern, it is necessary that temporal constraints hold.

5.3.2 Type for Binary Motion Pattern

In the theory of programming languages, a type is defined as a set of values. A binary motion pattern is represented by a grammar G_{bmp}. This grammar

generates a set of strings. Each string is an instance of a motion pattern between two objects. Therefore, we can treat a binary motion pattern as a type. In programming languages, types can be constructed in a bottom up way, starting from low level primitives and then gradually moving up. This is referred to as type induction. We use three type induction operators, namely, $|$, $*$ and $\bullet$. The $|$ operator expresses union, $*$ means Cartesian product and $\bullet$ is for concatenation. For explaining how types can be constructed out of values of binary qualitative relations, let us assume that a motion pattern is represented using n number of motion parameters. R_1, R_2, . . . R_n are the sets of JEPD binary qualitative relations, such that the set R_i represents the *i-th* movement parameter. The relation values for set R_i is denoted as r_{i1}, r_{i2}, . . . r_{im}. We present a grammar in BNF notation for constructing a type corresponding to a single primitive pattern:

$< btype > ::= R_1 \mid R_2 \mid . . . \mid R_n$

$R_1 ::= r_{11} \mid r_{12} \mid . . . \mid r_{1m}$

$R_2 ::= r_{21} \mid r_{22} \mid ... \mid r_{2l}$

.

.

.

$R_n ::= r_{n1} \mid r_{n2} \mid ... \mid r_{nk}$

$< basictypeexpression > ::= < btype >$

$< basictypeexpression > ::= < btype > * < btype >$

The syntactic category $< basictypeexpression >$ is an abbreviation for basic type expression and $< btype >$ is an abbreviation for basic type. From the above definition, we can see that the set of values in a basic type is same as the set of values for some R_i. A basic type expression is obtained by taking a Cartesian

product of basic types. The operator $*$ here is used to denote Cartesian product. Therefore, a $< \mathit{basictypeexpression} >$ is the language counterpart of a primitive motion pattern.

We introduce a syntactic category $< \mathit{bmptypeexpression} >$ for constructing a type corresponding to a binary motion pattern. The following BNF grammar shows how a type can be constructed for a binary motion pattern:

$< \mathit{bmptype} > ::= < \mathit{basictypeexpression} >$

$< \mathit{bmptype} > ::= < \mathit{bmptype} > \bullet < \mathit{bmptype} >$

$< \mathit{bmptype} > ::= < \mathit{bmptype} > \mid < \mathit{bmptype} >$

Here, $< \mathit{bmptype} >$ corresponds to a binary motion pattern type. A $< \mathit{basictypeexpression} >$ corresponds to a type for a single primitive pattern. The first line of the BNF grammar specifies that a binary motion pattern can be a single primitive pattern. The second line expresses the fact that a binary motion pattern can be obtained by taking concatenation of primitive patterns. The $\bullet$ operator is used to denote concatenation operation. The third line introduces the union operator. In a program, an object in the pattern is treated as a variable. Therefore, there should be language constructs that allow to declare a variable to be of a binary motion pattern type. The syntax for declaring a variable type is given below:

type $< \mathit{bmp_name} > = < \mathit{bmptype} >$

var $< \mathit{object_name} >:< \mathit{bmp_name} >$

In the above declaration, **type** is a keyword. A binary motion pattern is given a name and the syntactic category $< \mathit{bmp_name} >$ specifies this name. $< \mathit{bmptype} >$ denotes a binary motion pattern type. In the second line, the object with name $< \mathit{object_name} >$ is declared to be of type $< \mathit{bmptype} >$. Both, $< \mathit{bmp_name} >$ and $< \mathit{object_name} >$ are identifiers. Here, **var** is a keyword.

An Example

Let us assume that qualitative direction and distance are chosen as movement parameters. Let Q be the set of qualitative direction relations as defined in section 3.1. The set of distance relations is $D = \{veryclose, close, near, far\}$. Then examples of some *basictypes* are $(Same)$, $(Same-)$, $(close)$, (far) etc. Basic type expression is a set of ordered pairs such that each ordered pair belongs to the set $Q \times D$. Examples of *basic type expressions* are $(Same, close)$, $(Same-, near)$, (lr, far) etc. Type expressions can be constructed by using the concatenation operator and the union operator. Example of $<$ *bmptype* $>$ can be $(Same, close)\bullet(lr, far)$, $(Same, close)|(lr, far)$ etc. We declare below a binary motion pattern type and a variable *MoveTogether* to be of that type.

type $MoveTogether = (Same, Close)(Same-, Close)(Same, veryclose)$;

var X:*MoveTogether*;

The concatenation operator is not explicitly shown. Declaration of the reference object is not shown in the example.

5.3.3 Type for Basic Multi Object Pattern

A basic multiobject object pattern can also be considered as a set of strings. In a **BMOP**, we keep binary motion patterns between a number of primary-reference pairs. The set of strings in a **BMOP** can be obtained by taking a union of the strings belonging to each constituent binary motion pattern. In **QDL**, we introduce a type for **BMOP**. It is important to mention here that a **BMOP** does not have any hierarchy inside. This means that a **BMOP** does not contain any **BMOP** inside it. Hierarchy within a **BMOP** creates difficulty in its learning. Hierarchy is about grouping objects conceptually and specifying the temporal relationships between these groups. There are many possible ways of grouping objects in a **BMOP** conceptually. This perception of forming conceptual groups and creating a hierarchy of these groups is related with human cognition. The learning algorithm has no particular way of determining this hierarchy automatically. Since a **BMOP** does not contain a hierarchy, its corresponding type in

QDL does not allow nesting. Syntax for a **BMOP** type in *BNF* notation is given below:

BMOP < *bmop_name* >([< *type_par_list* >]) {

< *object_list* >

[< *reference* >]

< *bmp_declarations* >

< *var_declarations* >

< *temporal_constraints* >

< *max_hold_time_constraints* >

}

The objects in the basic multiobject object pattern are generated using the non terminal < *object_list* >. This can be done as:

< *object_list* > ::= **Object** < *id_list* > | **Object** < *type_par_list* >

< *id_list* > ::= < *object_id* >; | < *object_id* >,< *id_list* >

< *id_list* > ::= ;

< *object_id* > ::= < *identifier* >

< *type_par_list* > ::= < *type_parameter* >; | < *type_parameter* >,< *type_par_list* >

< *type_parameter* > ::= < *identifier* >

The reference can be declared using the syntax:

< *reference* > ::= **Reference** < *object_id* >;

< *object_id* > ::= < *identifier* >

The semantics of this declaration is that the identities of the objects that participate in the **BMOP** are recorded and associated with the name of the basic multiobject object pattern. The basic multiobject object pattern type is given a name and the the identifier < *bmop_name* > specifies this name. The identifier < *reference* > specifies the name of the reference. The non-terminal < *bmp_declarations* > generates the declarations of types corresponding to constituent binary motion patterns. We have already defined a non-terminal < *bmptype* > using which types for binary motion patterns can be defined. Therefore, the following productions take care of the declarations of types corresponding to constituent binary motion patterns:

< *bmp_declarations* > ::= **type** < *bmp_typename* > = < *bmptype* >; |

type < *bmp_typename* > = < *bmptype* >; < *bmp_declarations* >

< *bmp_typename* > ::= < *identifier* >

In the productions listed above, **type** is a keyword. Each binary motion pattern type is assigned a type name by the non-terminal *bmp_typename*. The semantics of these declarations is that each binary motion pattern type will be stored as a sequence of terminals and all these types will be associated with the basic multi object pattern under which they are defined.

We need to declare variables that are of binary motion pattern types. Each variable is related with exactly one object. The association of objects with variables can be implicit or explicit. For an explicit association, we need to state the name of the object that a variable is related with. In an implicit association, the name of a variable has to be same as the name of its corresponding object. In **QDL**, we have used an implicit association. The non terminal < *var_declarations* > generates the variable declarations:

< *var_declarations* > ::= **var** < *variable_name* > : < *bmp_typename* >; |

var < *variable_name* > : < *bmp_typename* >; < *var_declarations* >

< *variable_name* > ::= < *object_id* > | < *type_parameter* >

Temporal relationships between binary motion patterns is an important aspect. We have modeled these relationships using Allen's interval algebra. The interval algebra relations are specified between *HoldInterval*s of two binary motion patterns. Each interval algebra relation is represented in **QDL** by a binary operator. *HoldInterval* is a property of a binary motion pattern. Since a variable is declared to be of a binary motion pattern type, in language design, we can treat *HoldInterval* to be a property of a variable. Syntactically, the *HoldInterval* is written after a dot put at the end of a variable.The *BNF* grammar for temporal constraints is presented below:

$< temporal_constraints > ::= < variable_name >.< HI > < Allen_Operator >$
$< variable_name >.< HI >; \mid$

$< variable_name >.< HI > < Allen_Operator > < variable_name >.< HI >; < temporal_constraints >$

$< variable_name > ::= < object_id > \mid < type_parameter >$

$< Allen_Operator > ::= \mathbf{p} \mid \mathbf{pi} \mid \mathbf{m} \mid \mathbf{mi} \mid \mathbf{o} \mid \mathbf{oi} \mid \mathbf{d} \mid \mathbf{di} \mid \mathbf{s} \mid \mathbf{si} \mid \mathbf{f} \mid \mathbf{fi} \mid \mathbf{eq}$

The interval algebra operator **p** corresponds to the interval algebra relation *precedes*, **pi** corresponds to its inverse and so on. Each *HoldInterval* is denoted by two integers, the first for the start time point of the interval and the second for the end time point of the interval. Semantically, these temporal constraints are part of a **BMOP** and these constraints need to be stored along with the definition of a **BMOP**.

The maximum hold times for binary motion patterns are defined using the non-terminal $< max_hold_time_constraints >$. The maximum duration of time for which each constituent binary motion pattern may hold is specified. The maximum hold time for the entire **BMOP** can be computed from these individual maximum hold times of the constituent BMPs. The corresponding *BNF* productions will be like:

$< max_hold_time_constraints > ::= < max_hold_bmp >$

< *max_hold_bmp* > ::= < *variable_name* >.MaxHoldTime = (< *integer* >,< *integer* >); | < *variable_name* >.MaxHoldTime = (< *integer* >,< *integer* >); < *max_hold_bmp* >

< *variable_name* > ::= < *object_id* > | < *type_parameter* >

A **BMOP** type can be parameterised. The type parameters are place holders that can appear in the body of a **BMOP** definition. When the **BMOP** type is instantiated, these type parameters are replaced by actual arguments passed in instantiation. Each type parameter is an identifier and list of type parameters is a list of identifiers separated by commas.

Semantics of BMOP Definition

A **BMOP**, declared in a **QDL** program, can be mapped to a training record defined in section 4.3.1. This can be done using a syntax directed translation scheme. Productions of the non-terminal < *object_list* > fill the *Object_List* and *No_Of_Objects* fields of a training record. The non-terminals < *bmp_declarations* > and < *variable_name* > fills the *Object_Id* = *Pattern_Specification* field. It is important to note that the *HoldInterval_Start* and *HoldInterval_End* fields will be computed during parsing of < *max_hold_time_constraints* >. Once this mapping from **BMOP** definition to training record format is done during parsing, learning algorithm can be used on the training record so that the **BMOP** definition can be converted to its representational form described in section 4.5.1. In the representational form, the value of the *Temporal_Constraint_List* field is computed using a syntax directed scheme with the productions of the non-terminal < *temporal_constraints* >. The last two fields in the representation i.e. *MaxHoldTime_bmop* and *MaxHoldTimeList_bmp* are computed using productions of the non-terminal < *max_hold_time_constraints* >. Therefore, it is possible to convert a **BMOP** definition to the representational form introduced in section 4.5.1.

5.3.4 Type for Multi Object Pattern

In language design, we introduce a type for a multi object pattern (to be abbreviated as **MOP**). A multi object pattern can contain a hierarchy inside. When we write a program segment for specification or query, we can specify this hierarchy. In defining **BMOP**, we did not allow hierarchy because it is the basic multi object pattern that we wish to learn automatically from training data and the learning program presented in the thesis does not have the ability to determine hierarchy without human intervention. Automatic detection of event hierarchy is still an open research problem and some work is reported in [135] and also in [136]. In this work, a constraint based graph mining technique is used to discover a partonomy of classes of sub graphs corresponding to event classes. As we have not handled this issue in our work, we have chosen to use two different constructs **BMOP** and **MOP** in our language design. A multi object pattern type simply allows us to group objects and to specify temporal constraints between these groupings. A group inside a multi object pattern type may be some already defined **BMOP** or some already defined **MOP**. Combination of **BMOP** and **MOP** is also possible. We enforce the restriction that a type must be declared before its use. Syntax for a multi object type is given below:

MOP < *mop_typename* > {

< *var_decl_list* >

< *temp_constraints* >

}

< *var_decl_list* > ::= < *bmop_var_list* > | < *mop_var_list* >

< *bmop_var_list* > ::= **var** < *bmop_declaration* >

< *bmop_declaration* > ::= < *var_name* > : < *bmop_typename* >; |

< *var_name* > : < *bmop_typename* >; < *bmop_var_list* >

< *mop_var_list* > ::= **var** < *mop_declaration* >

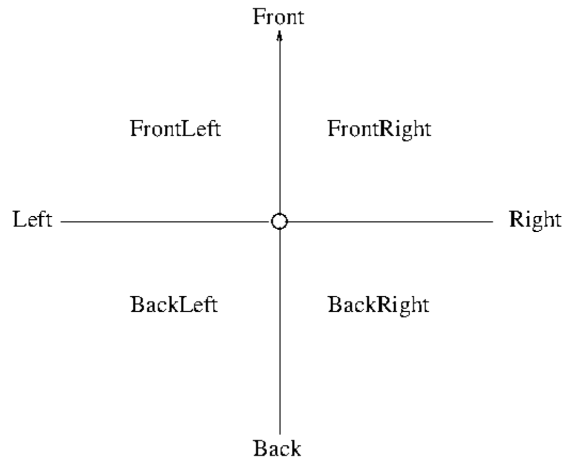

Figure 5.1: An Orientation Model for Point Objects

$< mop_declaration > ::= < var_name > : < mop_typename >; \mid$

$< var_name > : < mop_typename >; < mop_var_list >$

$< temp_constraints > ::= < var_name >.\mathrm{HI} < Allen_Op > < var_name >.\mathrm{HI};$
$\mid$

$< var_name >.\mathrm{HI} < Allen_Op > < var_name >.\mathrm{HI}; < temp_constraints >$

$< Allen_Op > ::= \mathbf{p} \mid \mathbf{pi} \mid \mathbf{m} \mid \mathbf{mi} \mid \mathbf{o} \mid \mathbf{oi} \mid \mathbf{d} \mid \mathbf{di} \mid \mathbf{s} \mid \mathbf{si} \mid \mathbf{f} \mid \mathbf{fi} \mid \mathbf{eq}$

$< var_name > ::= < identifier >$

A **MOP** is used to create hierarchies among conceptual grouping of objects and to express temporal constraints among these groups. A **MOP** declaration starts with a declaration of variables. A variable may be of type **BMOP** or **MOP**. Next comes an enumeration of temporal constraints between constituent **BMOP**s and **MOP**s. In expressing temporal constraints, **BMOP**s and **MOP**s are referred to by the corresponding variables. In a MOP definition, one may specify the maximum hold time for the MOP. Since the MOP can be nested inside other MOPs, the programmer may sometimes need to express the maximum hold time of the MOP in absolute terms. This maximum hold time is necessary because a **MOP** can be nested inside another.

Semantics of MOP Definition

Like a **BMOP**, a **MOP** definition can also be converted into a representational form during parsing. A representational form for a a **MOP** has a simple form where we need to store temporal constraints between constituent **BMOP**s or **MOP**s. This representational form can be like:

```
Pattern_Name
Variable_List
Temporal_Constraint_List
Maximum_Hold_Time
```

Variable_List holds the names of the constituent **BMOP**s and **MOP**s. The *Temporal_Constraint_List* is a enumeration of values where each value can be form the form:

```
(Pattern AllenOperator Pattern)
```

Here, *Pattern* can be a **BMOP** or a **MOP**. The *Maximum_Hold_Time* filed stores the maximum hold time of the multi object pattern. The value of this field can be computed from the maximum hold times of the constituent **BMOP**s or **MOP**s.

5.4 Execution Phases of QDL

A program, written in **QDL**, has different interpretations. Firstly, such a program can be used to query the presence of certain patterns in input data. In this case, we expect a "yes/no" answer after the execution of the program depending on whether the specified pattern is detected or not. Secondly, such a program can be used to store user defined type definitions as part of language definition. These stored types can be used as built-in types in writing programs. Thirdly, it is possible to learn a motion pattern from input data and represent it in the form of a **QDL** program. The different execution phases of a **QDL** program are shown in the Figure 5.2.

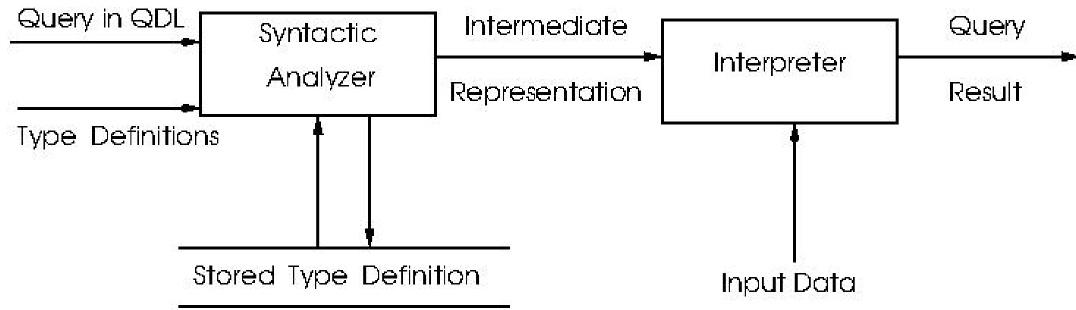

Figure 5.2: Execution Phases of QDL

The *Syntactic Analyzer* block checks for syntactic errors and it produces an intermediate representation for a syntactically correct program. The program may use built-in type definitions that are stored as part of language definition. The *Syntactic Analyzer* parses each **BMOP** and **MOP** in the source program using syntax directed translation schemes. If syntactically correct, then **BMOP**s and **MOP**s are transformed into their representational forms. Therefore, the *intermediate form* refers to the representational forms of the **BMOP**s and **MOP**s. The *Interpreter* takes this intermediate representational form and checks for the presence of the defined pattern in input data. The *Interpreter* uses algorithm 5 to recognise basic multi object patterns in the program. A **BMOP** defined in a **QDL** program has the same representational form as the one learned from training data. However, it is important to mention that only one **BMOP** is recognised among the learned **BMOP**s;but in **QDL** definition, multiple **BMOP**s can be recognised in a program. Once the processing status of the defined **BMOP**s are known i.e. it is known whether each **BMOP** is recognised or not, the *Interpreter* can analyse the conceptual groupings defined through **MOP**s. Since a **MOP** representation has information about constituent **BMOP**s and **MOP**s, processing can be done in a bottom up manner in order to check for temporal constraints. This means that we should start at lowest level of nesting for **MOP**s and then move higher up for resolving the temporal constraints. If the **MOP** at the highest level is recognised, then the entire motion pattern expressed by the **QDL** program is also recognised.

If the **QDL** program is a specification of types, then there is no need for any interpretation over input. The *Syntactic Analyzer* simply stores the type

definitions as built-in types after successful parsing.

The intermediate representation of a **BMOP** is obtained after successful parsing of a **QDL** program. It should contain all the details embodied in the program. Such an intermediate form for a **BMOP** is given below:

```
Program_Element=BMOP
Name=...
Object_List=...
Binary_Motion_Pattern_List=(object,pattern_string),(object,pattern_string)...
Reference=...
Temporal_Constraints_List=...
Maximum_Hold_Time_BMOP=...
```

Here, the name of the **BMOP** is stored along with a list of symbolic object names. These object names are declared inside the **BMOP**. In an enumeration of binary motion patterns, each pattern is stored along with the name of the object for which the pattern defines a type. The *Reference* field is optional. Its value can be *None* to indicate that no reference is used;otherwise it will contain name of the object that is declared as reference. Temporal constraint list gives a listing of Allen relations that hold between variables. The last field i.e. the maximum hold time for the **BMOP** is optional. Its value can be *None* to indicate that inside the definition, no such maximum hold time was indicated for the **BMOP**. Otherwise, the time is recorded as an ordered pair where the first element indicates the start time and the second indicates the finish time for the interval. We feel that sometimes, a programmer may need a mechanism to express the maximum time for which a **BMOP** may hold in absolute terms. This is the reason for including this in the syntax.

Intermediate representation for a **MOP** can be like:

```
Program_Element=MOP
Name=...
Object_List=(object,type,type_name),(object,type,type_name)...
```

```
Temporal_Constraints_List=...
Maximum_Hold_Time_MOP=...
```

Inside a **MOP**, variables can be of type **BMOP** and **MOP**. The object list gives enumerates the objects along with their types and type names. This *type* value can be either **BMOP** or **MOP**. By *type name*, we mean name of the **BMOP** or **MOP**. Temporal constraint list stores the Allen relations between pairs of **BMOP**s and **MOP**s. For a **MOP** also, we have allowed a specification of maximum hold interval in absolute terms and accordingly a field to store it in the intermediate form.

5.5 Example

We discuss an example program for specification of a motion pattern using **QDL**. Object declarations have global scope. The example is about movement of four persons mentioned in chapter 1 and shown in Figure 1.1. We create two conceptual groupings. In one group, we place the two persons who are walking along the street and approaching each other from opposite direction. In the other group, we place the persons crossing the road from opposite direction. Among the persons walking along the street, we assume the identity of the person facing the camera as A and that of the other person as B. In the other group, the person crossing the road from right to left is assigned the identity C. The other person, crossing the road from left to right, is assigned the identity D. We choose to abstract the objects as points. Spatial orientation, direction and distance are taken as movement parameters. The spatial orientation model is shown in Figure 5.1. The direction of movement of the object, abstracted as a point, is shown by the arrow head. Frame of reference is egocentric and the direction of movement sets up the *Front*. Other spatial orientation labels are assigned accordingly. Program below is one possible representation of this movement pattern:

BMOP Group1(α,β) {

Reference β;

type walk_along= (opposite, FrontLeft, far) (opposite, FrontLeft, near) (opposite, Left, near);

var α: walk_along;

}

BMOP Group2(α,β,γ) {

Reference γ;

type walk_across_rl= (FrontRight, rl, (far | near)) (Front, rl, near) (FrontLeft, rl, near) (FrontLeft, rl, far);

type walk_across_lr = (FrontLeft, lr, far) (FrontLeft, lr, near) (Front, lr, near) (FrontRight, lr, near) (FrontRight, lr, far);

var α: walk_across_lr;

var β: walk_across_rl;

}

MOP Example {

var g1:Group1(B,A);

var g2:Group2(C,D,A);

g2.HI **p** g1.HI;

}

Scope Rules

In a **QDL** program, there are two types of variables. Variables represent objects that participate in a motion pattern. Moreover, variables may also represent **BMOP** or **MOP**. The scope of *object variables* is global. Therefore, all such variables must be unique across the program. Variables for patterns have local

scope i.e. their scope is limited to the **BMOP** or **MOP** within which they are declared. In a **QDL** program, there are three kinds of type declarations. These are for **BMP**, **BMOP** and **MOP**. Type names for **BMOP** and **MOP** have global scope i.e. such a type name is visible in the entire program. Type names for **BMP** have local scope confined to the **BMOP** within which the type is defined. **QDL** does not support nested type definitions. A type must be defined before its use. Since all types are defined at the same level, the scope rule for resolving a type reference is very simple. Whenever a type name for a **BMOP** or **MOP** is encountered, the *Syntactic Analyzer* tries to resolve it using the type definitions encountered before.

In this chapter, we have presented the design of certain features of **QDL**, a qualitative description language for motion pattern representation and recognition. Programs written in **QDL** can express patterns that we want to recognise in input data stream. Otherwise, motion patterns can be learned and represented in the form of a **QDL** program. In the next chapter, we have shown how the standard taxonomy, outlined in chapter 2, can be represented as program segments in **QDL**. Moreover, a **BMOP** is learned and represented from video data. In the **QDL** representation of this **BMOP**, the productions of the learned grammar have been shown in Appendix A.