Dolce Language Specification

author(s) : ATOS

version : 3

status : Draft

source : Atos

document date : 13 December 2014­

­

­

owner : Atos Research & Innovation - Smart Objects Lab

© Copyright 2012, Atos España SAE All rights reserved. Reproduction in whole or in part is prohibited without the prior written consent of the copyright owner. For any questions or remarks on this document, please contact Atos, +34 91 214 8455.

­­­Contents

1 Preface 5

2 Introduction 6

3 Language Introduction 7

3.1 Getting Started 7

3.2 Event Attributes and Filtering 7

3.3 Comments and Arithmetic Expressions 8

3.4 Detecting Complex Events 9

3.5 Time Window (Sliding Time Back Window) 9

3.6 Tuple Window (Sliding Tuple Back Window) 12

3.7 Complex Event Functions 12

3.8 External Variables 13

4 Types, Operators and Expressions 15

4.1 Identifier Names 15

4.2 Data Types 15

4.3 Constants 15

4.4 Summary 18

4.5 Complex Event Functions 19

5 Program Structure 22

5.1 External Variables 22

5.2 Events 22

5.3 Complex Events 23

5.4 Comments 25

6 System Interface 26

6.1 File Name Extensions 26

6.2 Data Types 26

Figures

[Figure 4 – Dynamic behaviour of Complex Event Functions 24](#_Toc406411690)

[Figure 5 – Evaluation of the “After” Operation 31](#_Toc406411691)

[Figure 6 – Evaluation of the "During" Operation 31](#_Toc406411692)

[Figure 7 – Event Period Evaluation of the "Or" Operation 32](#_Toc406411693)

[Figure 8 – Event Period Evaluation of the "And" Operation 32](#_Toc406411694)

List of changes

| Version | Date | Description | Author(s) |
| --- | --- | --- | --- |
| 3 | December, 13, 2014 | 3rd version | Atos |
|  |  |  |  |
|  |  |  |  |

­­­­

# Preface

This document describes the Dolce language specification, version 3.

Dolce stands for “Description Language for Complex Events”.

It is designed for the SOL/CEP Complex Event Processing engine, created by the Smart Objects Lab, part of ARI Research & Innovation, a division of Atos Spain S.L.

Dolce is © 2014 Atos S.A. – All rights reserved.

# Introduction

The goals when designing the Dolce language have been twofold. First, it is conceived as a declarative language, which means that no programming expertise is needed. On the other hand, an attempt has been made to make it versatile for a number of real life situations. This includes built-in types that deal with location and temporal awareness, as well as basic functions such as counting and averaging.

# Language Introduction

The aim of this chapter is to give a brief introduction to Dolce, without entering too much in details or trying to be complete or precise. It is a simplified view of the world, but the idea is to get basic notions about the language, its purpose and how it relates to the CEP.

Concepts that may not seem to be immediately clear are explained in detail starting from Chapter 4.

## Getting Started

The following minimal Dolce program detects an event called *TemperatureReading*

**event** TemperatureReading

{

}

An event is declared using the event keyword, followed by an identifier stating the name of the event which the CEP must detect.

The previous program tells the CEP to consume and interpret the events. However, it is of no use, since it does not have any knowledge about complex events. The following code adds a complex event called *HeatWave*, by means of the complex keyword.

**event** TemperatureReading

{

}

**complex** HeatWave

{

}

Still, the CEP will not generate any complex events. For this to happen, it must first detect at least one event. The association is established using the detect statement.

**event** TemperatureReading

{

}

**complex** HeatWave

{

**detect** TemperatureReading;

}

This Dolce program will generate a *Heatwave* complex event each time a *TemperatureReading* comes in.

From the previous examples it can be seen that the statements within the events and complex events are always enclosed in { and } braces.

## Event Attributes and Filtering

The *TemperatureReading* event currently specified is of little use since it does not have any associated attributes that would allow the CEP to evaluate whether a heat wave situation has arisen.

The event can be taught about these attributes as follows, as shown in this code fragment:

**event** TemperatureReading

{

**use**

{

**int** SensorId,

**int** Temperature,

};

}

The use keyword tells the CEP that the event has the three listed attributes, each of the indicated type, which are int and time, respectively. The Dolce language is not explicit about the interpretation of their values – in the example, it is assumed that the Temperature attribute indicates the temperature in degrees Celsius, not Fahrenheit.

Now that the CEP knows about the event’s attributes, it is possible to apply a filter on them. The accept statement allows the CEP to selectively discard events and only let through the one it is interested in. In order to only allow temperature readings from sensor number 17, the code can be extended as shown below.

**event** TemperatureReading

{

/\*

These are the attributes that are used.

\*/

**use**

{

**int** SensorId,

**int** Temperature,

};

**accept** { SensorId == 17 }; // only accept events from sensor #17

}

## Comments and Arithmetic Expressions

The previous example also introduces some new concepts, such as comments and arithmetic expressions.

Comments are marked between the /\* and \*/ sequences. They can span multiple lines. One-line comments are also allowed by preceding them with the // sequence. All text after this sequence is ignored until a new line starts.

An arithmetic expression can be simple, such as the one to detect the specific sensor, but can also be more complicated.

Arithmetic operators include + – / and \*, for addition, subtraction, division and multiplication, respectively.

The expression 5 \* (6 ‑ 2) resolves to 20. As can be noted, the ( and ) brackets are used to group sub-expressions; the multiplication operator has precedence over the subtraction operator. The same expression without the braces, 5 \* 6 ‑ 2 will resolve to 28.

## Detecting Complex Events

Now that the event carries more useful information in the form of attributes, it’s time to revisit the detect statement.

This statement not only instructs the detection of complex events, it can also be used to condition the detection. In this case, a heat wave complex event is raised when the temperature exceeds 36 degrees Celsius.

**event** TemperatureReading

{

**use**

{

**int** SensorId,

**int** Temperature,

};

**accept** { SensorId = 17 };

}

**complex** HeatWave

{

**detect** TemperatureReading

**where** Temperature > 36;

}

The example shows how the where-clause is used to condition the detection. It also shows how the attributes declared in the *TemperatureReading* event are referenced. The Temperature attribute refers directly to the *TemperatureReading*. SensorId could also have been referenced.

## Time Window (Sliding Time Back Window)

In order to detect a heat wave, it is not sufficient to just raise a complex event whenever the temperature exceeds a certain value – there could be some hot days during a period that generally records moderate temperatures. To establish a definition of what is a heat wave, one could poorly define it to be a “period where the temperature exceeds 36 degrees Celsius during at least three days”.

The expression “during the last three days” is called a *sliding time window*. It is a window that starts from the current point in time stretching back a defined number of time units.

**event** TemperatureReading

{

**use**

{

**int** SensorId,

**int** Temperature,

};

**accept** { SensorId = 17 };

}

**complex** HeatWave

{

**detect** TemperatureReading

**where** Temperature > 36

**in** [ 3 **days** ];

}

The in statement defines a sliding time window of three days and always refers to a period starting from the current point in time back to, and including, three days ago.

Besides days, other temporal units are hours, minutes, seconds, years and months, together with their linguistic singular counterpart’s day, hour, minute, etc.

Thus, the same sliding time window could have been specified with [ 72 hours ] or [ 4320 minutes ].

**The above example is very simple; in fact, the behavior of the sliding windows has no effect for the publisher. In order to see the effect of sliding window we should include an arithmetic operation like average. The following example uses average operation.**

**event** TemperatureReading

{

**use**

{

**int** SensorId,

**int** Temperature,

};

**accept** { SensorId = 17 };

}

**complex** HeatWave

{

**detect** TemperatureReading

**where avg(**Temperature) > 36

**in** [ 3 **days** ];

}

In this case, each time an event arrives the CEP, It takes into account that the average must be calculated with the last Temperature value and the values that are in the window time into the past.

The following figure shows the events taken in account if three events arrived before now:



## Tuple Window (Sliding Tuple Back Window)

Instead of measuring elapsed time, once could also measure the number of *occurrences* of a particular event.

**event** TemperatureReading

{

**use**

{

**int** SensorId,

**int** Temperature,

};

**accept** { SensorId = 17 };

}

**complex** HeatWave

{

**detect** TemperatureReading

**where avg(**Temperature) > 36

**in** [3];

}

The example detects complex events during the last three temperature readings. Dolce calls these “last three temperature readings” a *sliding tuple window*. In fact, due to its nature, a tuple window is always “sliding”, therefore it can also be referred to as *tuple window*. A tuple window is specified like a sliding time window, omitting the temporal specifier.

Since the tuple window always refers to only one event from the detect statement, namely *TemperatureReading*, it can also we written as:

**in** [3];

The name is implied from that used in the detect statement. However, this notation is not allowed if more than one event where involved in the detection, because the CEP would not know for which event to follow the occurrences.

## Complex Event Functions

Assuming temperature readings are received at reasonable intervals at different times during a day, the previous temperature example does not take into account the fact that the temperature usually varies during the 24 hours that make up that day. A sustained reading of the temperature specified in the complex event will mostly never occur, and thus the complex event will never be raised.

When improving the definition to “a period where the *average* temperature exceeds 20°C during at least three days”, the complex event could be specified as follows:

**complex** HeatWave

{

**detect** TemperatureReading

**where avg**(Temperature) > 20

**in** [ 3 **days** ];

}

The **avg** function is what is called a *complex event function*. Complex event functions only make sense when used in combination with a time window or a tuple window.

Dolce has other complex event functions, which are:

|  |  |
| --- | --- |
| count | counts occurrences of events |
| sum | reflects the total result of all the values of particular event attribute |
| diff | calculates the relative difference between two expressions or an event value |

The CEP ensures that the results calculated by the functions constantly reflect the values of the event attributes as time or tuples pass through the sliding window. This means that attribute values of the events that slide out of the window are removed from the evaluation.

To illustrate this, consider the *SmogAlert* complex event that is raised when more than 1000 cars pass through a certain zone within 60 minutes. The *TrafficSensorReport* event contains an attribute indicating the number of cars that have passed through mentioned zone.

**event** TrafficSensorReport

{

**use**

{

**int** SensorId,

**int** NumberOfCars,

**int** SmogLevel

};

**accept** { SensorId = 17 };

}

**complex** TrafficJam

{

**detect** TrafficSensorReport

**where sum**(NumberOfCars) > 1000

**in** [ 60 **minutes** ];

}

Assuming that at time **T (now)** 350 cars have passed, and at **T- 59minutes** another 680 cars had passed, the complex event will be raised, because the count *at that point in time* exceeds 1000 (1030 to be exact).

However, supposing that no further cars pass for the next couple of hours, at **T+12** the result of sum will be 680, because the value of 350 recorded at time **T** is no longer part of the sliding window.

As time goes by, at some point the result of 680 will also slide out of the window and the result in sum will be zero.

## External Variables

The Dolce language provides a feature that accepts external variables that can be transmitted to the CEP at runtime. These external variables can be used to influence the behavior of the events and complex events, without having to modify their definitions.

**external** **int** MAX\_TRANSACTIONS = 400;

**external int** WITHDRAWAL\_AMOUNT = 5000;

**external duration** MEASUREMENT\_PERIOD = 5 **hours**;

As can be seen, an external variable is declared by using the external keyword, followed by the type and name of the variable and a mandatory default value. As a convention, external variables are spelled in capital letters.

Suppose that the traffic authority wants change the conditions under which the traffic jam alert is raised, it could do this by substituting certain values with external values.

**external** **int** MINIMUM\_CARS\_FOR\_JAM = 1000;

**external duration** PERIOD = 1 **hour**;

**complex** TrafficJam

{

**detect** TrafficSensorReport

**where sum**(NumberOfCars) > MINIMUM\_CARS\_FOR\_SMOG

**in [** PERIOD ];

}

# Types, Operators and Expressions

## Identifier Names

Names of identifiers are made up of letters and digits, and must start with a letter. The underscore character is considered as a letter. Letters may be in both upper and lower case.

MyEvent, \_the\_event and event123\_ are examples of valid identifier names.

3Event is not a valid identifier name, neither is &3.

Keywords such as event, complex, days, minute, detect and accept cannot be used as identifier names.

The Dolce language is case sensitive. Thus myEvent is different from MyEvent.

## Data Types

The Dolce language recognizes the following data types:

|  |  |
| --- | --- |
| int | A signed integer value, whose range is limited by the architecture on which the CEP is deployed. |
| float | Floating point number, whose range is limited by the machine architecture on which the CEP is deployed. |
| string | 0-terminated string of 8-bit characters. |
| duration | Period of certain length. |

Table 1 – Data Types

## Constants

When declaring external variables and complex event payloads, they can be assigned values of a certain type, for example:

**complex** MyAlert

{

**payload**

{

**int** SensorId = 2334,

};

}

**external** **int** MINIMUM\_CARS\_FOR\_SMOG = 1243;

### Numerical Constants

These constants apply to the int and float data types.

*Integer constants* (int) are declared by specifying them as sequence of digits, optionally preceded by a minus sign.

Examples of valid integer constants are:

1 2321 -231445 34834002

Invalid integer include:

92UL 56L 100.0

*Float constants* (float) are defined as an optional minus sign followed by a sequence of one or more digits, followed by a dot symbol, followed by one or more digits. The dot symbol is mandatory to denote a float constant – otherwise it is interpreted as an integer. Valid float constants are:

0.444 -0.23 -43.2823

Examples of invalid float constant are:

12 -45 23.4e43 .34 93.

### Duration

The duration type is used to specify a time period of a specific length.

It is expressed as:

*time-units* *unit-type*

The time units are expressed as an integer value, which can be of a different number of types.

Supported unit types are:

|  |  |
| --- | --- |
| second | seconds |
| minute | minutes |
| hour | hours |
| day | days |
| week | weeks |
| month | months |
| year | years |

Table 3 – Duration Unit Types

Note that the singular and plural versions of each specifier have the same meaning and can be used interchangeably.

Examples of duration constants are:

1 day

2 days

1 months

2 months

30 year

35 years

0.5 hour // ERR: Use 30 seconds

It is not allowed to specify time units in fractions. Half hours must be expressed in minutes.

### Relational and Logical operators

The supported relational operators are <, <=, > and >=, all of the same precedence. The represent the less-than, less-or-equal, greater-than and greater-or-equal operations.

Of lower precedence are the equality operators == and !=, for equality and non-equality, respectively.

Finally, the operators && and || are used for logical and- and or-operations.

### Event Operators

Events and complex events can be compared using the and or operators. These operators only apply to the detect statement for complex events. They are explained in more detail in section 5.3.3.

### Event Attributes

Event attributes are specified with the use-keyword in the event declaration and can be used in the detect statement of a complex event declaration.

### Associativity and Precedence

*Precedence* refers to the priority with which the operators are treated. Operators with a higher precedence are evaluated before those with a lower precedence.

*Associativity* refers to the order in which the operands of the expression are evaluated in case there are multiple operators with the same precedence.

In the Dolce language, the associativity is from left to right. The precedence is specified in the overview below. Operators that are grouped together have the same precedence and are evaluated according to associativity.

| Operator | Description |
| --- | --- |
| < <= > >= | Less, less than, greater, greater than |
| == != | Equal, non equal |
| && | Logical and |
| || | Logical or |

Table 4 – Operator Precedence

Precedence and associativity can be overridden by placing parts of the expression in ( and ) brackets.

Duration

## Complex Event Functions

Complex event functions are built-in methods that provide statistical information about the event attributes involved in the complex event.

### Count

The count function calculates the number of events that have occurred. It is the only function that does not operate on event attributes, but rather on the event itself. It has the following syntax

"count" "(" *event-name* ")"

The function returns a value of type int.

### Sum

This function contains the total of the event attributes values. It has the form

"sum" "(" *event-attribute-name* ")"

It can only be used with event attribute of type float or integer.

The function returns a value of type float.

### Average

This function contains the average of the event attributes values. It has the form

"avg" "(" *event-attribute-name* ")"

It can only be used with event attribute of type float or integer.

The function returns a value of type float.

### Difference

This function calculates the relative difference between two event attribute values, as a percentage fraction of 100.

It has two forms. The first form

"diff" "(" *event-attribute-name* ")"

Calculates the difference between the latest and the last available event attribute value in the time window or tuple window.

The second form

"diff" "(" *expression1* "," *expression2* ")"

calculates the difference between the two expressions, relative to the time window or tuple window.

Valid expressions are variable declarations, event attributes and constants; all must be of type int or float.

A common usage would be comparing the value of different instances of the same event – for example the latest and the penultimate. This can be achieved using the Window Offset Specifiers described in section 4.4.4.

The following example calculates the percentage difference between the latest attribute value and that of the attribute farthest away in time relative to the time or tuple window:

diff ( TickerEvent.stockPrice@@, TickerEvent.stockPrice )

If the latest attribute value is 38 and the farthest attribute value is 18, the function will yield 111.1.

Both functions can only be used with attributes or expression of type float or int; they can also only be used in conjunction with a time or tuple window.

Both functions return a value of type float.

### Automatic Updating

The result of many of the functions is at all times dependant on the time or tuple windows. As the tuples come in, or as time goes by, events become part of the window at one end, and disappear from the window on the other end, resulting in an ever-changing configuration of values that determine the final result of the complex event functions.

This is best illustrated in the following diagram, where a time window of 4 time units is applied to a stream of incoming events, which, for the purpose of the example come in at intervals synchronized to the time unit of the sliding time window.

The figures in the box refer to the values associated to one of the event attributes and that are used as input for the complex event functions. Empty boxes mean that no event was captured in the time frame. The running result of each of functions is shown for the time window as it is at time t and time t + 2.

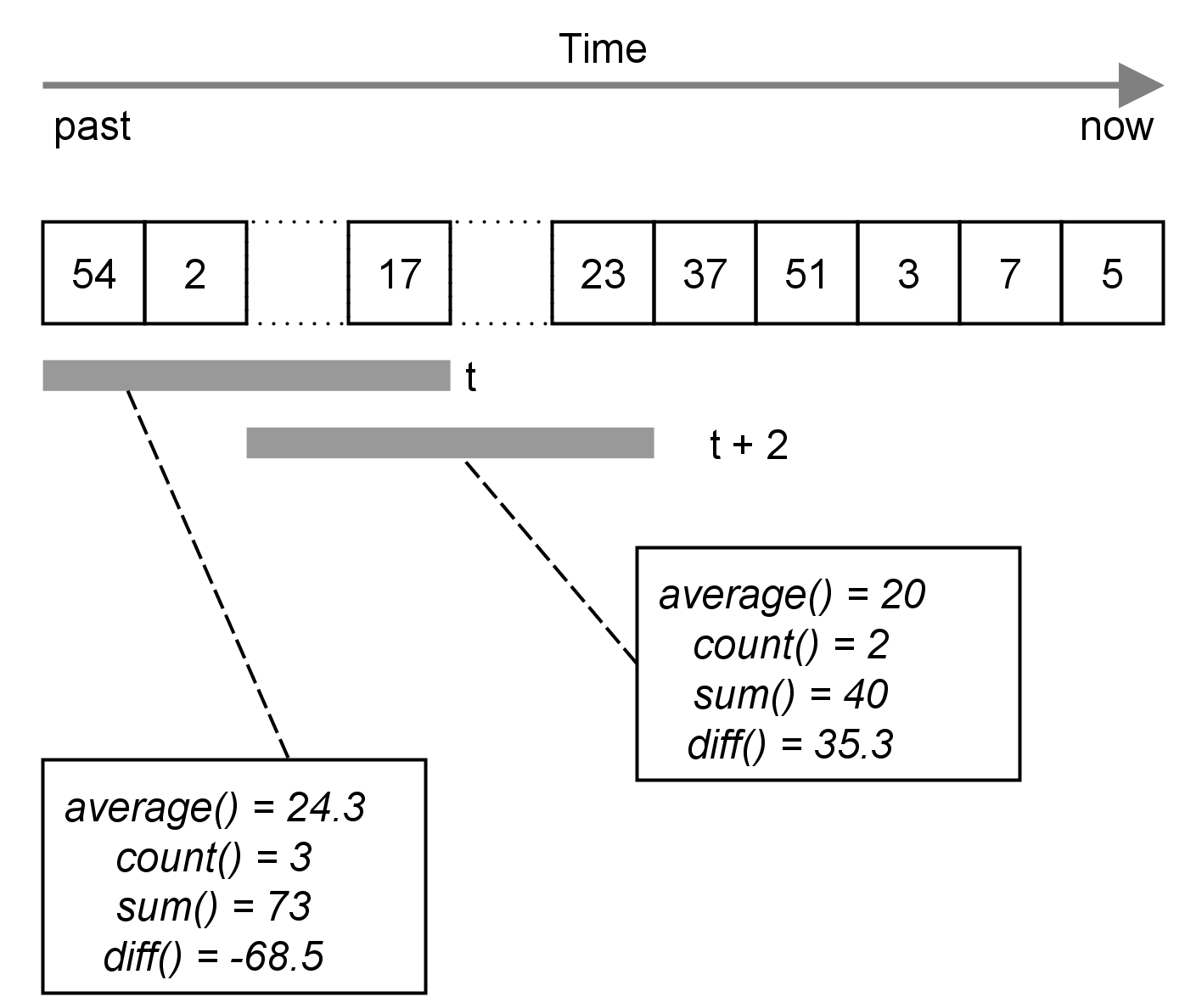


Figure 4 – Dynamic behaviour of Complex Event Functions

Note that not all Complex Event Functions are continuously updated – exceptions is the two-parameter variant of the diff() function.

### Summary

The following table summarizes the functions

| Function | Return Type | Description | Automatic Updating |
| --- | --- | --- | --- |
| count(event) | int | Counts events | Yes |
| sum(event-attribute) | float | Sums event attribute values | Yes |
| avg(event-attribure) | float | Averages event attribute values | Yes |
| diff(event-attribute) | float | Calculates percent difference between event attribute values | Yes |
| diff(expr1, expr2) | float | Calculates percent difference between two expressions | No |

Table 6 – Complex Event Functions

# Program Structure

This chapter deals with the structure and format of a valid Dolce program. Essentially, a Dolce program declares events and the complex events that will use them to detect certain situations. Events are distinguished from each other by a unique name, as are complex events. Two or more events or complex event must not have the same name.

The events, complex events and externals can be declared in any order, although a best practice is to put the external at the beginning, followed by the events and ending with the complex events.

## External Variables

External variable declarations have the form:

"external" *variable-type* *variable-name* "=" *constant-expression*

The external variables form a link to the world outside the CEP. They can be changed at runtime and thus influence the behavior of the processing.

Valid examples are:

external int MAX\_TRANSACTION = 1000;

external duration = 10 hours;

The Dolce language does not specify the mechanism of changing the value of the external variables, nor the behavior of the CEP at the time the values are changed.

## Events

An event has the form

**"**event" *event-name*

"{"

*Statements and declarations*

"}"

An event may have no statements at all, so that a minimal event can be declared as:

**"**event" *event-name* "{" "}"

An event declaration consist of use and accept statements and the event period declaration. Each statement or declaration may only appear once. They are separated using a semicolon.

The event name must follow the rules for identifier names as specified in section 4.1.

### The Use Statement

This statement enumerates the attributes that are present and used in the external event.

The reason why it is called a “use” statement is that it tells the CEP to only use those attributes from the external event. The external event my contain many more attributes. It is not possible to introduce attributes that are not present in the external attribute.

The use statement has the form:

"use" "{"   
 *attr-type1 attr-name1* "," *attr-type2* *attr-name2*, ...   
"}"

The attribute type is one of the valid types specified in section 4.2. Attribute names follow the same conventions as the event name.

Attributes may not be repeated and do not necessary follow the order in which they are specified in the external event.

All the attributes that are used by the complex events referring to the event must be declared in the event.

### The Accept Statement

This statement allows the filtering of incoming events, by applying an expression on one or more of the attributes.

It is denoted as follows:

"accept" "{" *expression* "}"

Any attribute names used in the expression must have been declared using the use statement. The expression can have any form. This means that expressions can be specified that impede the event from being processed, such as the expression 1 == 2.

The use of complex event functions (section 3.7) is not allowed.

Events that do not have an accept statement, are accepted by default.

## Complex Events

A complex event has the form

"complex" *complex-event-name*

"{"

*declarations and statements*

"}"

A complex event may not have any content, which means that a minimal complex event can be declared as:

"complex" *complex-event-name* "{" "}"

However, an empty complex event will be of little use.

The possible definitions and statements for the complex event are the payload definition, the group , detect and lasts statements. They are separated using a semicolon.

### The Detect Statement

This statement is fundamental to the detection of complex events. It has the form

"detect" *logical-event-expression*

[ "where" *expression* ]

[ "in" *time-window or tuple-window* ]

The sequence "where" *expression* is called the where-clause, the sequence "in" *expression* is called the window-clause. Both are optional.

The next sections deals with the individual components that make up the detect-statement.

### Logical Event Expressions

The logical-event-expression is an enumeration of one or more events, combined with logical operators, which eventually evaluates to True or False. The table below lists the expressions and their behavior.

|  |  |
| --- | --- |
| !E | Event E must not occur. |
| E and F | Both event E and event F must occur. |
| E or F | Either event E or event F must occur. |
| E after F | Event E must happen after event F. |
| E during F | Event E must happen during event F. |

Table 7 – Logical Event Expressions

It is important to note that in this context, the detect statement can combine *both events and complex events*. This means that a complex event can be an aggregate of other complex events.

The first expression in the above table is evaluates to True if the logical event expression has *not* occurred – since this only makes sense when using a time window or tuple window, it is mandatory to add an window-clause to the detect statement.

Precedence

The rules for precedence and associativity explained in section 4.4.5, also apply to the evaluation of the expression for detect events and complex events. The table below summarizes the precedence.

|  |  |
| --- | --- |
| ! | Absence operator. |
| and | Event E must happen after event F. |
| or | Event E must happen during event F. |

Table 8 – Precedence of Logical Event Operators

For example, the expression A after B and C will first evaluate if event (or complex event) A happens after B, because after has a higher precedence then and. When the condition is true, it will check if both happen in conjunction with C.

On the other hand, the expression A after (B and C) will first evaluate if A happens in conjunction with C, then if both happen after A. The and precedence is imposed by placing (B and C) in brackets.

Likewise, the expression A after B after C is evaluated as follows: the first condition that must come true is “A occurs after B”. When this is the case, then an evaluation will be made if both come after C.

Like precedence, associativity can be overridden using brackets: A after (B after C)will first evaluate if B occurs after C.

### Where-clause

The optional Where-clause can contain expressions that further condition the complex event. Expressions are explained in chapter 4.

### Window-clause

The Window-clause determines the window during which events are detected. Two types of windows exist: time windows and tuple windows.

Sliding Time Windows

Within the time window category, two different types of time windows can be distinguished – *sliding time windows* that measure the time always from the current moment stretching back to a certain amount of time in the past, and *fixed time windows*, that measure a particular point in time during a certain number of time units.

A *sliding time window* can be expressed as indicated below:

"[" *duration* "]"

The duration constant is explained in section 4.3.4. The duration may also be specified as an identifier referencing an external variable *of type duration*.

Examples of time windows are:

[ 13 days ]

[ 29 days ]

[ 7 months ]

[ 2 months ]

[ 15 year ]

[ 91 years ]

[ DURATION\_1 ]

Tuple Windows

Tuple windows are similar to time windows, but instead of referring to the time that has passed, they refer to the number of occurrences of a particular event. Their notation is:

"[" *tuple-count* [ *event-name* ] "]"

The event name does not have to be specified when only one event is part of the detect statement; it is implied from the event being detected. If however the detect statement applies to more than one event, the tuple window must include the name of the event for which the occurrences must be tracked. Only one event name can be specified. Also, only one tuple window can be specified per complex event.

The tuple count may be specified using an identifier representing an external variable *of type int*.

The following are examples of valid tuple windows:

[ 42 TrafficNotifications ]

[ 5 LoginAttempts ]

[ 15 ]

[ LOGIN\_COUNT LoginAttempts ]

Some invalid tuple windows are:

[ 92 hours ]

[ 701 4435 ]

[ MyCount ]

## Comments

The Dolce source code can contain comment sections that start with /\* and end with \*/. Within the comment section, no expression of the type /\* or \*/ is allowed.

It is also possible to specify line comments, that start with // and span the entire line until the newline character.

# System Interface

The Dolce language specification does not make assumptions about the format in which events are presented to the CEP. It only specifies the variables that are expected to be present in the events and those that are emitted by the complex events.

It relies on software that maps the external event into an internal format that matches both the declared Event in Dolce and the expected internal format.

Likewise, Complex Events are expected to be translated from the internal format of the CEP to an external format, depending on the expected usage.

The translations are out of scope for this document.

## File Name Extensions

Although the specification does not mandate the mechanism by which the complex event processing definitions are transmitted to the CEP, it is not uncommon to place them in a file – Dolce language files are expected to have the “.dolce” extension.

## Data Types

The specification does not mandate the internal representation of the data types, such as int or float. The choice is left to the CEP that implements the specification and will mostly depend on the hardware architecture. For example, integers could be represented using a 32 or 64 bits, or even as 16 bits, whereas the float data type may have single or double precision.

Appendix A

This section lists the reserved Dolce keywords.

accept

and

avg

char

complex

count

day

days

detect

diff

duration

event

external

float

group

hour

hours

in

int

minute

minutes

month

months

or

second

seconds

string

sum

time

use

until

valid

week

weeks

where

year

years