Position Metamodel Reference Documentation

Version 2023-09-12

Contents

Chapter 1

Introduction

The position language is a language to represent position information, for source tracking. A position is represented as a continuous region in a textual source (the *source text*). This language is typically not used by itself, but other languages can use this language to provide position information storage. By using a common language for position information, the position information can be handled generically by e.g. parsers, type checkers and text editors.

The position language is part of the common functionality provided by the Eclipse ESCET™ project [2].

In this report, the position language is defined. The position language is defined using a so-called *conceptual model*, also known as *metamodel* by the Object Management Group (OMG). A metamodel represents concepts (entities) and relationships between them. The position metamodel is described using (Ecore) class diagrams [1], where classes represent concepts, and associations represent relationships between concepts. Static semantic constraints and relations that cannot be represented using class diagrams are stated in the class documentation of the metamodel. The metamodel and the accompanying constraints are used primarily to formalize the syntax of the internal (implementation) representation of the language.

This report is organized as follows. The notations and conventions used in this document are explained in Chapter 2 and Chapter 3 describes the position metamodel.

Chapter 2

Notations and conventions

2.1 Ecore class diagrams

Metamodels are represented using Ecore class diagrams, which are very similar to UML class diagrams. In Ecore class diagrams, *classifiers* represent concepts, and *associations* represent relationships between concepts. There are two kinds of classifiers, namely *data types* and *classes*.

Data types are used for simple types, whose details are not modeled as classes. Data types are identified by a name. Examples of data types include booleans, numbers, strings (optionally restricted using regular expressions), and enumerations.

A class is also identified by its name, and can have a number of structural features, namely attributes and references. Classes allow *inheritance*, giving them access to the structural features of their supertypes/basetypes.

Attributes are identified by name, and they have a data type. Associations between classes are modeled by *references*. Like attributes, references are identified by name and have a type. However, the type is the class at the other end of the association. A reference specifies lower and upper bounds on its multiplicity. The multiplicity indicators that can be used are shown in Table 2.1. Finally, a reference specifies whether it is being used to represent a stronger type of association, called *containment*.

Graphically, data types are depicted as rectangles. The rectangles have a yellow background. The data type name is shown at the top inside the rectangle. The Java class name is shown

Table 2.1: Multiplicity indicators

Indicator	Meaning
n	Exactly n (where $n \geq 1$), default notation
$n..n$	Exactly n (where $n \geq 1$), alternative notation
$n..m$	n up to and including m (where $n \geq 0$, $m \geq 1$, and $m > n$)
$n..*$	n or more (where $n \geq 0$)

Table 2.2: Ecore diagram classifier icons

Icon	Meaning
	Data type
	Enumeration
	Class
	Abstract class

Table 2.3: Ecore diagram feature icons

Icon	Meaning
	Attribute with multiplicity [0..1]
	Attribute with multiplicity [1..1]
	Reference with multiplicity [0..1]
	Reference with multiplicity [1..1]
	Reference with multiplicity [0..*]
	Reference with multiplicity [1..*]

below it. Enumerations differ slightly. They have a green background. Instead of the Java class name, the enumeration literals are listed below the name of the enumeration.

Classes are depicted as rounded rectangles with a yellow background. The class name is shown at the top inside the rectangle. Abstract classes have a grey background, and the class name is shown in italic font. The names, types and multiplicity of the attributes are shown inside the rectangle. References for which the target class is not part of the diagram, are listed as well. Features from base classes are listed using a grey font.

Tables 2.2 and 2.3 shows the various icons used in Ecore class diagrams for classifiers and features.

Inheritance relations are depicted as arrows between two classes with a (non-solid) triangle on the side of the superclass. A reference is depicted as an arrow between two classes, labeled with its name and its multiplicity. A containment reference is depicted with a solid diamond at the side of the containing class.

2.2 Metamodel documentation conventions

Each (sub-)package is described in a separate section. An informal description of the package is followed by the Uniform Resource Identifier (URI) of the package, the namespace prefix, and a list of all the direct sub-packages. All classifiers defined in the package are described in subsections. First the data types are described, then the enumerations, and finally the classes. The data types are ordered lexicographically, as are the enumerations and classes.

For data types, an informal description of the data type is followed by the name of the data type,

the instance class name, basetype, and the (regular expression) pattern.

For enumerations, an informal description of the enumeration is followed by information about the enumeration literals, which are ordered lexicographically. For each enumeration literal, a short informal description is included. The default value of the enumeration (the default literal), is indicated as well.

For classes, an informal description of the class is followed by the inheritance hierarchy. Note that all classes that do not have an explicit supertype in the Ecore, implicitly inherit from *EObject*[1]. Therefore, all the inheritance hierarchies start in *EObject*. The inheritance hierarchies are followed by a listing of all the directly derived classes of the class. Finally, all the structural features of the class are listed, including the inherited ones. The structural features of the supermost type are listed first, and the ones of the actual class are listed last. Secondary ordering is lexicographical.

For each structural feature, the type is indicated ('attr' for attributes, 'ref' for references, and 'cont' for containment references). This is followed by the name of the structural feature, the multiplicity, a colon, and the type. If the structural feature is inherited from a supertype, that is indicated as well. Finally, an informal description of the structural feature is provided.

---

[1]Actually, in the implementation, *org.eclipse.emf.ecore.EObject* and all classes from metamodels are interfaces. Implementation classes implement the interfaces and have names ending in *Impl*. E.g. *org.eclipse.emf.ecore.impl.EObjectImpl* implements *org.eclipse.emf.ecore.EObject*.

Chapter 3

Position metamodel

The position metamodel consists of only one packages, the *position* package. The class diagram is presented and described in the section below.

3.1 Package position

Figure 3.1 shows the *position* package.
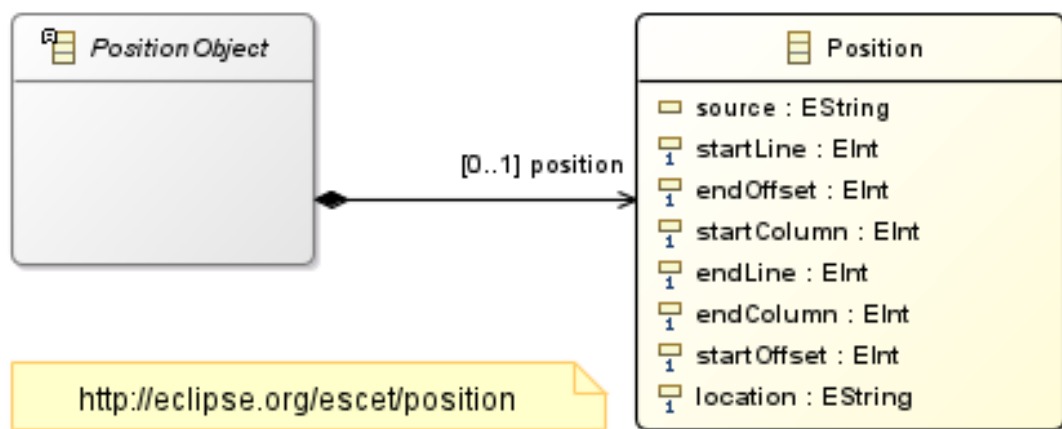
Figure 3.1: *position* package

The position package contains classes used to represent position information, for source tracking. A position is represented as a continuous region in a textual source (the *source text*). Positions are expressed as offset from the start of the text and as line and column pair. Lines are delimited by a line-feed codepoint (U+000A).

The *Position* (Section 3.1.1) class represents actual position information. The abstract *Posi-*

tionObject (Section 3.1.2) class can be used as a base class for other classes, and allows those classes to store position information.

Package URI http://eclipse.org/escet/position

Namespace prefix position

Sub-packages none

3.1.1 Position (class)

Position (source tracking) information.

Constraints:

- **Position.lines** The `startLine` must be smaller than or equal to the `endLine`.

- **Position.columns** If the `startLine` is equal to the `endLine`, the `startColumn` must be smaller than or equal to the `endColumn`.

- **Position.offsets** The `startOffset` must be smaller than or equal to the `endOffset`.

EObject
└ *Position*

Direct derived classes: none

attr **endColumn** [1] : *EInt*
    The 1-based codepoint index at the `endLine` of the end (inclusive) of the position region with respect to the start of the line.
    Constraints:

    - **Position.endColumnValue** Value must be greater than or equal to one.

attr **endLine** [1] : *EInt*
    The 1-based line index of the end (inclusive) of the position region with respect to the start of the source text.
    Constraints:

    - **Position.endLineValue** Value must be greater than or equal to one.

attr **endOffset** [1] : *EInt*
    The 0-based codepoint index of the end (inclusive) of the position region with respect to the start of the source text.
    Constraints:

    - **Position.endOffsetValue** Value must be greater than or equal to zero.

attr **location** [1] : *EString*
  The location of the source file that contains the position. Must be an absolute local file system path, with platform specific file separators. The path does not have to refer to an existing file. That is, it may not be assumed that a file with that path actually exists on disk.

attr **source** [0..1] : *EString*
  Optional source identification. Usually, this is a file name.

attr **startColumn** [1] : *EInt*
  The 1-based codepoint index at the `startLine` of the start (inclusive) of the position region with respect to the start of the line.

  Constraints:

  - **Position.startColumnValue** Value must be greater than or equal to one.

attr **startLine** [1] : *EInt*
  The 1-based line index of the start (inclusive) of the position region with respect to the start of the source text.

  Constraints:

  - **Position.startLineValue** Value must be greater than or equal to one.

attr **startOffset** [1] : *EInt*
  The 0-based codepoint index of the start (inclusive) of the position region with respect to the start of the source text.

  Constraints:

  - **Position.startOffsetValue** Value must be greater than or equal to zero.

3.1.2   PositionObject (abstract class)

Base class for other classes, facilitating the storage of position information.

EObject
  ∟ *PositionObject*

Direct derived classes: none

cont **position** [0..1] : *Position*
  Optional position information.

Chapter 4

Legal

The material in this documentation is Copyright (c) 2010, 2023 Contributors to the Eclipse Foundation.

Eclipse ESCET and ESCET are trademarks of the Eclipse Foundation. Eclipse, and the Eclipse Logo are registered trademarks of the Eclipse Foundation. Other names may be trademarks of their respective owners.

License

The Eclipse Foundation makes available all content in this document ("Content"). Unless otherwise indicated below, the Content is provided to you under the terms and conditions of the MIT License. A copy of the MIT License is available at `https://opensource.org/licenses/MIT`. For purposes of the MIT License, "Software" will mean the Content.

If you did not receive this Content directly from the Eclipse Foundation, the Content is being redistributed by another party ("Redistributor") and different terms and conditions may apply to your use of any object code in the Content. Check the Redistributor's license that was provided with the Content. If no such license exists, contact the Redistributor. Unless otherwise indicated below, the terms and conditions of the MIT License still apply to any source code in the Content and such source code may be obtained at `http://www.eclipse.org`.

Bibliography

[1] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. Addison-Wesley, 2009.

[2] Contributors to the Eclipse Foundation. Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET). `https://eclipse.dev/escet`.