

CIF Metamodel Reference Documentation

Copyright (c) 2010, 2024 Contributors to the Eclipse Foundation

Version 2024-03-17

Contents

1	Introduction	7
2	Notations and conventions	8
2.1	Ecore class diagrams	8
2.2	Metamodel documentation conventions	9
3	CIF metamodel	11
3.1	Scoping	12
3.1.1	Reference wrapping	12
3.1.2	Scoping rules	13
3.1.3	Visibility	15
3.2	Package cif	17
3.2.1	CifIdentifier (datatype)	17
3.2.2	InvKind (enumeration)	17
3.2.3	SupKind (enumeration)	19
3.2.4	AlgParameter (class)	19
3.2.5	ComplexComponent (abstract class)	20
3.2.6	Component (abstract class)	21
3.2.7	ComponentDef (class)	22
3.2.8	ComponentInst (class)	23
3.2.9	ComponentParameter (class)	24
3.2.10	Equation (class)	25
3.2.11	EventParameter (class)	26
3.2.12	Group (class)	27
3.2.13	Invariant (class)	29

3.2.14	IoDecl (abstract class)	30
3.2.15	LocationParameter (class)	30
3.2.16	Parameter (abstract class)	31
3.2.17	Specification (class)	31
3.3	Package declarations	34
3.3.1	AlgVariable (class)	34
3.3.2	Constant (class)	36
3.3.3	ContVariable (class)	37
3.3.4	Declaration (abstract class)	39
3.3.5	DiscVariable (class)	39
3.3.6	EnumDecl (class)	42
3.3.7	EnumLiteral (class)	43
3.3.8	Event (class)	43
3.3.9	InputVariable (class)	44
3.3.10	TypeDecl (class)	45
3.3.11	VariableValue (class)	46
3.4	Package automata	47
3.4.1	Alphabet (class)	48
3.4.2	Assignment (class)	48
3.4.3	Automaton (class)	49
3.4.4	Edge (class)	52
3.4.5	EdgeEvent (class)	53
3.4.6	EdgeReceive (class)	54
3.4.7	EdgeSend (class)	55
3.4.8	ElifUpdate (class)	55
3.4.9	IfUpdate (class)	56
3.4.10	Location (class)	57
3.4.11	Monitors (class)	58
3.4.12	Update (abstract class)	59
3.5	Package types	60
3.5.1	BoolType (class)	60
3.5.2	CifType (abstract class)	61

3.5.3	CompInstWrapType (class)	61
3.5.4	CompParamWrapType (class)	62
3.5.5	ComponentDefType (class)	63
3.5.6	ComponentType (class)	63
3.5.7	DictType (class)	64
3.5.8	DistType (class)	64
3.5.9	EnumType (class)	65
3.5.10	Field (class)	65
3.5.11	FuncType (class)	66
3.5.12	IntType (class)	66
3.5.13	ListType (class)	67
3.5.14	RealType (class)	68
3.5.15	SetType (class)	68
3.5.16	StringType (class)	69
3.5.17	TupleType (class)	69
3.5.18	TypeRef (class)	70
3.5.19	VoidType (class)	70
3.6	Package expressions	71
3.6.1	BinaryOperator (enumeration)	71
3.6.2	StdLibFunction (enumeration)	74
3.6.3	UnaryOperator (enumeration)	77
3.6.4	AlgVariableExpression (class)	77
3.6.5	BaseFunctionExpression (abstract class)	78
3.6.6	BinaryExpression (class)	78
3.6.7	BoolExpression (class)	81
3.6.8	CastExpression (class)	81
3.6.9	CompInstWrapExpression (class)	82
3.6.10	CompParamExpression (class)	83
3.6.11	CompParamWrapExpression (class)	84
3.6.12	ComponentExpression (class)	84
3.6.13	ConstantExpression (class)	85
3.6.14	ContVariableExpression (class)	86

3.6.15	DictExpression (class)	87
3.6.16	DictPair (class)	87
3.6.17	DiscVariableExpression (class)	88
3.6.18	ElifExpression (class)	88
3.6.19	EnumLiteralExpression (class)	89
3.6.20	EventExpression (class)	89
3.6.21	Expression (abstract class)	90
3.6.22	FieldExpression (class)	91
3.6.23	FunctionCallExpression (class)	91
3.6.24	FunctionExpression (class)	92
3.6.25	IfExpression (class)	93
3.6.26	InputVariableExpression (class)	94
3.6.27	IntExpression (class)	95
3.6.28	ListExpression (class)	95
3.6.29	LocationExpression (class)	96
3.6.30	ProjectionExpression (class)	97
3.6.31	RealExpression (class)	98
3.6.32	ReceivedExpression (class)	98
3.6.33	SelfExpression (class)	99
3.6.34	SetExpression (class)	100
3.6.35	SliceExpression (class)	100
3.6.36	StdLibFunctionExpression (class)	101
3.6.37	StringExpression (class)	102
3.6.38	SwitchCase (class)	105
3.6.39	SwitchExpression (class)	106
3.6.40	TauExpression (class)	107
3.6.41	TimeExpression (class)	107
3.6.42	TupleExpression (class)	108
3.6.43	UnaryExpression (class)	108
3.7	Package functions	109
3.7.1	AssignmentFuncStatement (class)	110
3.7.2	BreakFuncStatement (class)	111

3.7.3	ContinueFuncStatement (class)	112
3.7.4	ElifFuncStatement (class)	112
3.7.5	ExternalFunction (class)	113
3.7.6	Function (abstract class)	114
3.7.7	FunctionParameter (class)	115
3.7.8	FunctionStatement (abstract class)	116
3.7.9	IfFuncStatement (class)	116
3.7.10	InternalFunction (class)	117
3.7.11	ReturnFuncStatement (class)	118
3.7.12	WhileFuncStatement (class)	119
3.8	Package cifsvg	120
3.8.1	SvgCopy (class)	120
3.8.2	SvgFile (class)	122
3.8.3	SvgIn (class)	122
3.8.4	SvgInEvent (abstract class)	123
3.8.5	SvgInEventIf (class)	124
3.8.6	SvgInEventIfEntry (class)	124
3.8.7	SvgInEventSingle (class)	125
3.8.8	SvgMove (class)	126
3.8.9	SvgOut (class)	127
3.9	Package print	128
3.9.1	PrintForKind (enumeration)	129
3.9.2	Print (class)	129
3.9.3	PrintFile (class)	131
3.9.4	PrintFor (class)	131
3.10	Package annotations	132
3.10.1	AnnotatedObject (abstract class)	133
3.10.2	Annotation (class)	133
3.10.3	AnnotationArgument (class)	134
3.11	Package position	134
3.11.1	Position (class)	135
3.11.2	PositionObject (abstract class)	137

4 Distributions	138
5 Legal	140
Bibliography	140

Chapter 1

Introduction

CIF is a declarative modeling language for the specification of discrete event, timed, and hybrid systems as a collection of synchronizing automata. The CIF tooling supports the entire development process of controllers, including among others specification, supervisory controller synthesis, simulation-based validation and visualization, verification, real-time testing, and code generation.

CIF is one of the tools of the Eclipse ESCET[™] project [3].

In this report, the CIF language is defined. The CIF 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 CIF metamodel is described using (Ecore) class diagrams [2], 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, Chapter 3 describes the CIF metamodel, and Chapter 4 describes the stochastic distributions in more detail.

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.

For the CIF language, arrows colored in red are containments, but should be interpreted semantically as references. That is, in order to allow wrapping expressions and types to be used, they are containments instead of references. See also Section 3.1.

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 sub-

sections. 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.

¹Actually, 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

CIF metamodel

The CIF metamodel consists of the following packages:

- *cif* package: includes component structure and component definition/instantiation (Section 3.2 at page 17),
- *declarations* sub-package: includes CIF declarations, including among others events and variables (Section 3.3 at page 34),
- *automata* sub-package: includes automata, the basic components of CIF models (Section 3.4 at page 47),
- *types* sub-package: includes CIF types (Section 3.5 at page 60),
- *expressions* sub-package: includes CIF expressions (Section 3.6 at page 71),
- *functions* sub-package: includes internal and external user-defined CIF functions (Section 3.7 at page 109),
- *cifsvg* sub-package: includes CIF/SVG I/O declarations (Section 3.8 at page 120),
- *print* sub-package: includes print I/O declarations (Section 3.9 at page 128),
- *annotations* sub-package: includes annotations (Section 3.10 at page 132), and
- *position* package: position information (Section 3.11 at page 134).

The class diagrams are presented and described in the respective sections below.

Note that the *position* package is not actually part of the CIF metamodel. It is actually part of the *position* metamodel, but it is used by the CIF metamodel to represent position information. As such, the documentation for the *position* package is included in this document, for completeness. All CIF classes directly or indirectly derive from *PositionObject* (Section 3.11.2).

3.1 Scoping

3.1.1 Reference wrapping

One of the features of CIF is that pretty much anything in the entire specification can be referenced from anywhere in the specification. This is however complicated by the existence of a component definition/instantiation mechanism.



Figure 3.1: Wrapping expression example: situation before.

Take a look at Figure 3.1a. It shows a CIF model with a component definition *A*, which contains a location *x*. This component definition is instantiated twice, once as *a1*, and once as *a2*. Finally, there is a component *b*, which refers to location *x* of component *a1*. The reference is written in the CIF textual syntax.

Figure 3.1b shows the corresponding instance of metamodel classes that is generated by the compiler to represent the *a1.x* text. Instead of just generating a reference to an instance of the *Location* (Section 3.4.10) class, an expression is used. An instance of the *LocationExpression* (Section 3.6.29) class is wrapped in an instance of the *CompInstWrapExpression* (Section 3.6.9) class, which also references an instance of the *ComponentInst* (Section 3.2.8) class.

What we see here is a direct representation of the textual syntax, as we reference location *x* via component instantiation *a1*. We need to keep track of references via component instantiations. If we were to only generate an instance of the *LocationExpression* (Section 3.6.29) class, or just use a direct reference to an instance of the *Location* (Section 3.4.10) class, we would no longer be able to distinguish between location *x* in component *a1*, and location *x* in component *a2*.

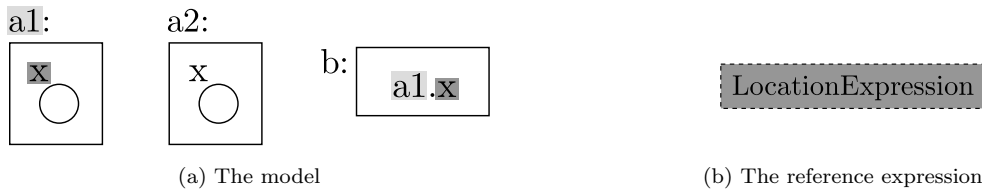


Figure 3.2: Wrapping expression example: situation after.

Figure 3.2a shows the model after elimination of component definition/instantiation. We see that components instantiations *a1* and *a2* are replaced by components *a1* and *a2*, which are essentially equal to the body of component definition *A*. As such, we now have two locations *x*, one in component *a1*, and one in component *a2*. The text to refer to location *x* of component *a1*, as used in component *b*, is still *a1.x*.

Figure 3.2b shows the corresponding reference expression. It is no longer necessary to use a

wrapping expression, as we can refer to location \mathbf{x} in component $\mathbf{a1}$ directly, as it is a real location, not one that we can get multiple versions of due to instantiation. As such, there is no confusion about which location we mean by that reference expression.

Note that wrapping does not only apply to references via component instantiations, but also to references via component parameters. Also, it does not just apply to expressions references, but also to type references. See also the *CompInstWrapType* (Section 3.5.3), *CompParamWrapType* (Section 3.5.4), *CompInstWrapExpression* (Section 3.6.9), and *CompParamWrapExpression* (Section 3.6.11) classes.

From this we can conclude that reference expressions, reference types, and their wrapping variants, are necessary to keep track of the component instantiations and component parameters via which we reference certain objects. That is, we need to correctly capture all of the information contained in the textual representations of references, in our metamodel representation. This is for instance needed to be able to correctly reroute references during the elimination of component definition/instantiation. After that elimination step however, wrapping expressions and wrapping types are no longer present. References do remain expressions, and as such, in the example, an instance of the *LocationExpression* (Section 3.6.29) is contained, instead of a direct reference to an instance of the *Location* (Section 3.4.10) class. Such containments are colored in red in the metamodel, as already mentioned in Chapter 2.

3.1.2 Scoping rules

As previously mentioned in Section 3.1.1, one of the features of CIF is that pretty much anything in the entire specification can be referenced from anywhere in the specification. This is however complicated by the existence of a component definition/instantiation mechanism, which caused a need for reference wrapping.

This section explains the scoping rules of CIF. Figure 3.3 shows an example CIF specification. The rectangles in the figure indicate components and component definitions. The outer rectangle (the dashed one), indicates the entire specification. The text fragments are written in a slightly simplified version of the CIF textual syntax. In the remainder of this section, the example is used to explain the scoping rules.

The first thing to note about the figure are the different colors. They indicate the different scopes of the specification. The outer-most scope (the red scope) is the specification scope. We walk the containment hierarchy downwards until we hit a component definition. From there, we start a new scope, and repeat the process. That is, specifications and component definitions start a new scope. The idea is that component definitions are definitions, and as such we can't refer to objects in component definitions directly. If we were to be able to reference objects in component definitions directly, and there would not be any instance of such a component definition, or there would be multiple instances, to what would we be referring? So, to solve this ambiguity, only once we instantiate a component definition, can we refer to objects within that definition (via component instantiation reference wrapping expressions for instance).

An example of this is component instantiation \mathbf{c} : $\mathbf{C}()$ in component \mathbf{a} . In component \mathbf{a} , in the red scope, we can't directly reference objects from component definition \mathbf{C} , as those objects are in the green scope. Once we have \mathbf{c} , we can use it to refer to objects in \mathbf{C} , such as $\mathbf{c.b}$. In fact, $\mathbf{c.b.a.a.x}$ refers to location \mathbf{x} (black scope) in component $\mathbf{a.a}$ (black scope) in component definition \mathbf{B} , via component instantiations \mathbf{c} (red scope to green scope) and \mathbf{b} (green scope to



Figure 3.3: Scoping and visibility example.

black scope).

Note how a component definition itself is part of the enclosing scope, while the parameters are part of the new scope that is introduced by the component definition. That is, the parameter cannot be referenced directly from the enclosing scope. If we look at component definition K, we see that component parameter `c` is part of the new blue scope. The type of component parameter `c` is component `C`, which is resolved in the enclosing scope (the red scope).

Within a single scope (single color in the figure), anything can be referenced directly. For instance, in component `a` (red scope), references `b.x` and `b.y` refer to locations in automaton `b`, which is part of the same red scope. Furthermore, component definition `P` (from component `i`) can be instantiated directly in component `a`, as both are in the red scope.

In general, anything within the same scope can be referenced directly, as can anything from ancestors scopes (the parent scope, the parent of the parent scope, etc). An example of the latter is `d.x` in an invariant of component `C.B.a` (in the black scope), which refers to location `C.d.x` of automaton `C.d`, in the green parent scope. The only two ways to reference things in component definitions that the reference itself is not a part of, are via component instantiations and via component parameters. For an example of the latter, see invariant `inv c.d.x` in component definition K.

Note that it is not allowed to instantiate component definitions which, in the instantiation context, can only be referenced via another component instantiation or via a component parameter. That is, it is not allowed to instantiate a component definition `X`, declared inside other component definition `Y`, from outside `Y`. Conceptually, in an instantiated component, there are no component definitions.

Also, it is not allowed to refer to a parameter of a component definition, via component instantiations or via other component parameters. Once again, conceptually, in an instantiated component, there are no parameters.

For user-defined functions, it is not allowed to refer to anything inside a function (parameters etc) from outside the function. The other way around, from inside a function, it is not allowed to refer to variables, locations, events, components, component definitions, etc. Objects with constant values (such as constants etc) may be referenced from inside functions, as may types (type declarations, enumerations, etc).

The scoping rules for tuple fields are special. See the *ProjectionExpression* (Section 3.6.30) class for further details.

3.1.3 Visibility

Within the metamodel, the scoping rules are enough. However, in the CIF textual syntax, the concept of *visibility* is important as well.

If we look at Figure 3.3, we see that relative references to objects that are ‘in scope’ are possible. For instance, in component `a`, there is a relative reference to `b.x`. Reference `c.b.a.a.x` is a relative reference as well, even though it uses some component instantiations to reach other scopes. Relative references refer to objects in the same sub-scope or deeper sub-scopes, possibly via references to other scopes.

Relative references are also possible to objects defined at higher levels, as for instance shown by

discrete variable `C.B.a.a.v0`, which has type `t`. This type `t` is defined one sub-scope above, in `C.B.a`.

The other discrete variables in that same automaton, discrete variables `v1` and `v2`, also have type `t` as their type, but refer to different types `t`. Since only one object with a given name can be visible in a scope or sub-scope, types `t` (in the specification scope) and `C.B.t` are not visible in automaton `C.B.a.a`. They are however ‘in scope’, as they are type declarations in the same scope or in ancestor scopes.

To be able to refer to objects that are in scope, but are *hidden* by other declarations of the same name, absolute references can be used. There are two variants: scope absolute references and specification absolute references. The type `.t` of discrete variable `v1` is a scope absolute reference. The reference starts with a single dot (`.`) and is then relative to the root of the current scope (the black scope). The type `^t` of discrete variable `v2` is a specification absolute reference. The reference starts with a caret (`^`) and is then relative to the root of the current specification (the red scope).

Note that functions, similar to components, introduce a sub-scope.

Note that tuple projection expressions may introduce a sub-scope. This scope has the tuple fields as objects. This kind of sub-scope is special in the sense that the fields are only in scope of the projection expression. Note that as for all scopes, fields may hide other objects with the same name, from parent scopes. Note that if it is a field reference, the textual reference to that field must be a single identifier. For instance, for a tuple-typed variable `t`, `t[i]` can be a field reference but `t[i + 1]` can not. If the index expression is a reference to another object that can be referred to by a single identifier, then that object may be hidden by a tuple field with that same name. In such cases the object can’t be referred to by a single identifier as the field name takes precedence, hiding the object. A more complex textual reference, such as a scope absolute reference, must then be used instead to refer to the object.

3.2 Package cif

Figure 3.4 shows the *cif* package. This package contains classes that are used to represent the component structure of CIF models. It also includes the component definition/instantiation related classes.

The *Component* (Section 3.2.6) class represents CIF components. There are three types of components: automata (*Automaton* (Section 3.4.3) class), groups (*Group* (Section 3.2.12) class), and component instantiations (*ComponentInst* (Section 3.2.8) class). Automata and groups share a common base class (*ComplexComponent* (Section 3.2.5)), since they share many features. The root object of every CIF model is an instance of the *Specification* (Section 3.2.17) class, a special form of *Group* (Section 3.2.12).

The bottom part of the *cif* package diagram deals with component definitions (*ComponentDef* (Section 3.2.7) class) and their parameters.

Package URI <http://eclipse.org/escet/cif>

Namespace prefix cif

Sub-packages *declarations* (Section 3.3), *automata* (Section 3.4), *types* (Section 3.5), *expressions* (Section 3.6), *functions* (Section 3.7), *cifsvg* (Section 3.8), *print* (Section 3.9), *annotations* (Section 3.10)

3.2.1 CifIdentifier (datatype)

A CIF identifier starts with a letter or an underscore character, and is followed by zero or more letters, digits, and/or underscore characters.

Note that it is allowed to use keywords from the textual syntax as identifiers. In the CIF textual syntax, this is allowed as well, but requires that the keyword is prefixed with a dollar sign (\$). This prefix is not allowed in the metamodel, as there is no ambiguity.

Name CifIdentifier

Instance class name java.lang.String

Basetype <http://www.eclipse.org/emf/2002/Ecore#EString>

Pattern [A-Za-z_][A-Za-z0-9_]*

3.2.2 InvKind (enumeration)

Invariant kind. Indicates whether an invariant is a state (exclusion) invariant or a state/event exclusion invariant, and for state/event exclusion invariants also in what form it is written.

literal **EventDisables**

A state/event exclusion invariant, giving a sufficient condition for an event to be disabled.

A predicate and event are given with the invariant. The event is disabled in states where

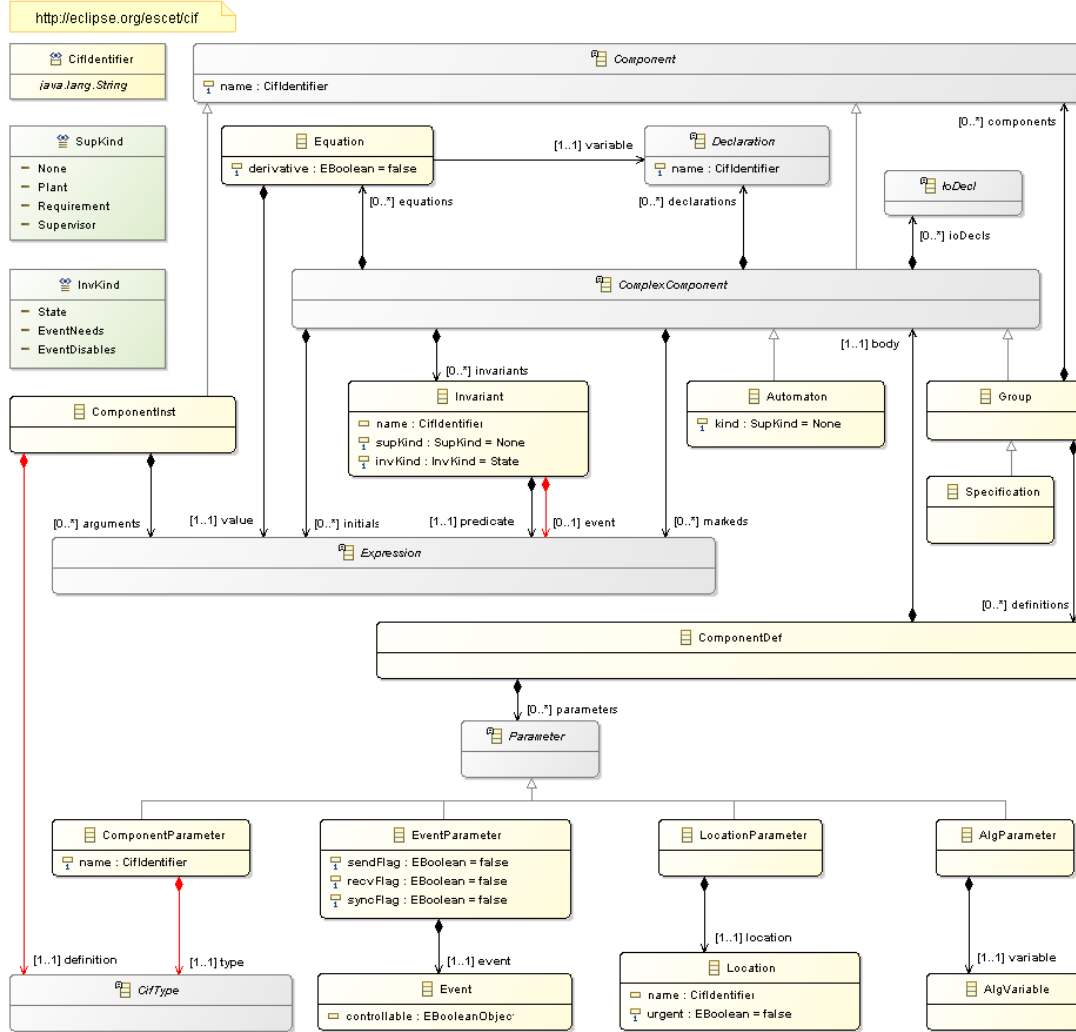


Figure 3.4: *cif* package

the predicate holds. If the predicate does not hold in a state, the event may or may not be enabled, depending on the automata and other (invariant) restriction of the system.

literal **EventNeeds**

A state/event exclusion invariant, giving a necessary condition for an event to be enabled. A predicate and event are given with the invariant. The event is only enabled in states where the predicate holds. More precisely, if the predicate does not hold in a state, the event is disabled.

literal **State** (default)

A state (exclusion) invariant. A predicate is given with the invariant that indicates which states are allowed. All states not satisfying the predicate may never be active states, i.e. may never be reached.

3.2.3 SupKind (enumeration)

Supervisory kind. Indicates how an automaton or invariant is to be interpreted for supervisory synthesis and related algorithms/tools.

literal **None** (default)

Regular automaton or invariant. Indicates that the kind is considered irrelevant (it is just an automaton or invariant, nothing else). This kind is usually used if the automaton or invariant is not to be used for synthesis, verification, etc.

literal **Plant**

Plant automaton or invariant. A plant automaton specifies the behavior of the uncontrolled physical system, or a part of it. Plant automata may abstract away from details of the actual physical system, if those details are considered irrelevant. A plant invariants restricts the behavior of the uncontrolled physical system, or a part of it.

literal **Requirement**

Requirement automaton or invariant. A requirement automaton or invariant (or requirement) represents the required behavior of the controlled system, or a part of it. Requirements can be synthesized, verified, etc.

literal **Supervisor**

Supervisor automaton or invariant. A supervisor automaton (or supervisor) can be used to control the uncontrolled plant. A supervisor invariant restricts the behavior of the supervised system, or a part of it.

3.2.4 AlgParameter (class)

An algebraic (variable) parameter of a *ComponentDef* (Section 3.2.7).

Note that there are no constant parameters. However, constants may be provided as arguments for algebraic parameters.

EObject

⊢ *PositionObject* (Section 3.11.2)

\sqsubset *Parameter* (Section 3.2.16)
 \sqsubset *AlgParameter*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

cont **variable** [1] : *AlgVariable*
 The algebraic variable declaration that represents the algebraic parameter.
 Constraints:

- **AlgParameter.noValue** The *variable* declaration must not have a value, since the argument should provide it.

3.2.5 ComplexComponent (abstract class)

Base class for more complex components. That is, for components that have declarations etc. This class was introduced to allow sharing of structural features between the *Automaton* (Section 3.4.3) and *Group* (Section 3.2.12) classes.

EObject
 \sqsubset *PositionObject* (Section 3.11.2)
 \sqsubset *Component* (Section 3.2.6)
 \sqsubset *ComplexComponent*

Direct derived classes: *Automaton* (Section 3.4.3), *Group* (Section 3.2.12)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

attr **name** [1] : *CifIdentifier* (inherited from *Component*)
 The name of the component.

Constraints:

- **Component.name (non-fatal)** Component names must not start with *e*-, *c*-, or *u*-, as those prefixes are reserved for events.

cont **declarations** [0..*] : *Declaration*
 The declarations of the component.

cont **equations** [0..*] : *Equation*
 The equations of the component.

cont **initials** [0..*] : *Expression*
 The initialization predicates of the component. A CIF model can only start in states that satisfy the initialization predicates.

If no predicates are given, the initialization predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between initialization predicates.

See the documentation of the derived classes for information on how to combine the initialization predicates with other levels.

Constraints:

- **ComplexComponent.initialTypes** The initialization predicates must have boolean types.

cont **invariants** [0..*] : *Invariant*

The invariants of the component. A CIF model can only be in a state if that state satisfies the invariants.

If no invariants are given, the invariant is *true*. If multiple invariant are given, this feature represents the logical conjunction of those invariants. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between invariants.

See the documentation of the derived classes for information on how to combine the invariants with other levels.

cont **ioDecls** [0..*] : *IoDecl*

The I/O declarations of the component.

Constraints:

- **ComplexComponent.maxOneSvgFile** Must not contain more than one *SvgFile* (Section 3.8.2) declaration.
- **ComplexComponent.maxOnePrintFile** Must not contain more than one *PrintFile* (Section 3.9.3) declaration.

cont **marked** [0..*] : *Expression*

The marker predicates of the component. Used as liveness property for, among others, supervisory controller synthesis.

If no predicates are given, the marker predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between marker predicates.

See the documentation of the derived classes for information on how to combine the marker predicates with other levels.

Constraints:

- **ComplexComponent.markedTypes** The marker predicates must have boolean types.

3.2.6 Component (abstract class)

Base class of all CIF components. Components model the structure of a CIF model.

EObject

⊑ *PositionObject* (Section 3.11.2)

⊑ *Component*

Direct derived classes: *ComplexComponent* (Section 3.2.5), *ComponentInst* (Section 3.2.8)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

attr **name** [1] : *CifIdentifier*

The name of the component.

Constraints:

- **Component.name (non-fatal)** Component names must not start with *e*-, *c*-, or *u*-, as those prefixes are reserved for events.

3.2.7 ComponentDef (class)

A component definition. Component definitions allow reuse of components. They can be instantiated using a *ComponentInst* (Section 3.2.8). Note that component definitions are not components themselves. They must be instantiated to become actual components.

In the metamodel, the *ComponentDef* class itself is just a wrapper that has parameters and a body. The name of the body defines the name of the component definition. The body is considered a part of the component definition, and as such, the body (component) may not be referenced directly.

If we eliminate a component definition, the instantiation is replaced by the body of the component definition, all reference parameters are substituted in the body (which has become an actual component), and algebraic parameters become local algebraic variables in that component (the body).

Constraints:

- **ComponentDef.uniqueDecls** The names of all parameters, as well as the declarations and invariants in the body of the component definition, must be unique within that component definition. For bodies that are automata, we have to take into account the names of all declarations (including locations) and invariants (including location invariants) of those automata. For bodies that are groups, we have to take into account the names of all declarations, child components, and component definitions, of those groups.

EObject

⊑ *PositionObject* (Section 3.11.2)

⊑ *ComponentDef*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **body** [1] : *ComplexComponent*
The body of the component definition.

Note that the body of a component definition is a *ComplexComponent* (Section 3.2.5), and not a *Component* (Section 3.2.6). As such, component instantiations are not allowed as body. This constraint is implied by the CIF textual syntax. Obviously, the body may *contain* a component instantiation, if the body is a group.

Constraints:

- **ComponentDef.selfReference** The body of a component definition may not, directly or indirectly, contain an instantiation of the component definition itself. That is, a component definition may not be defined in terms of itself.

cont **parameters** [0..*] : *Parameter*
The parameters of the component definition.

3.2.8 ComponentInst (class)

A component instantiation. Instantiates a *ComponentDef* (Section 3.2.7).

EObject

⊆ *PositionObject* (Section 3.11.2)
⊆ *Component* (Section 3.2.6)
⊆ *ComponentInst*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

attr **name** [1] : *CifIdentifier* (inherited from *Component*)
The name of the component.

Constraints:

- **Component.name (non-fatal)** Component names must not start with e_, c_, or u_, as those prefixes are reserved for events.

cont **arguments** [0..*] : *Expression*
The arguments to match each of the parameters of the component definition.

Constraints:

- **ComponentInst.argumentCount** The number of arguments of the component instantiation must be equal to the number of parameters of the component definition being instantiated.
- **ComponentInst.argumentTypes** The type of each argument must match the type of the corresponding parameter. That is:

- For each component parameter, the argument must be a reference to a matching component. That is, for each parameter with its component definition type, the argument must be a reference to a component that is an instance of that component definition.
- For each event parameter, the argument must be a reference to an event.
 If the parameter specifies the controllability of the event, the event used as argument must have the same controllability as the parameter. If the parameter does not specify the controllability, the controllability of the argument does not matter (it may have controllability, but it may also not have controllability).
 If the parameter specifies the type of the event, the event used as argument must have an equal type (including ranges). If the parameter does not specify the type of the event, the type of the event used as argument does not matter (it may have a type, but it may also not have a type).
 If the parameter specifies event usage restriction flags, the event used as argument must support all those usages. If the parameter specifies any flags, it requires that the event used as argument supports at least those usages. If the parameter does not specify any flags, it requires that the event used as argument supports all usages. If the event used as argument is an event declaration, it automatically supports all usages. If the event used as argument is an event parameter, its usage restriction flags should be checked. If the event used as argument does not have any flags, it supports all usages. If the event used as parameter does have one or more flags, it supports only those usages specified by the flags.
 It is not possible to use the ‘tau’ event as argument.
- For each location parameter, the argument must be a reference to a location.
- For each algebraic parameter, the argument must be an expression with a compatible type. If the parameter’s type has a range, the argument’s type must have one as well, and the range of the type of the argument must be entirely contained in the range of the type of the parameter.

cont **definition** [1] : *CifType*

The component definition to instantiate.

This feature contains *CifType* (Section 3.5.2) instances, to allow for wrapping types to be used. See also Section 3.1.

Constraints:

- **ComponentInst.compDefRef** The type must reference a component definition.
- **ComponentInst.compDefInScope** The component definition reference must satisfy the scoping rules.

3.2.9 ComponentParameter (class)

A component parameter of a *ComponentDef* (Section 3.2.7).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Parameter* (Section 3.2.16)

⊆ *ComponentParameter*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

attr **name** [1] : *CifIdentifier*

The name of the component parameter. Note that other parameters get their name via the contained declaration. Since the component parameter references a component definition, it does not contain such a declaration, and the name is instead included in this class.

Constraints:

- **ComponentParameter.name (non-fatal)** Component parameter names must not start with *e*-, *c*-, or *u*-, as those prefixes are reserved for events.

cont **type** [1] : *CifType*

The type of components that may be passed along as argument.

This feature contains *CifType* (Section 3.5.2) instances, to allow for wrapping types to be used. See also Section 3.1.

Constraints:

- **ComponentParameter.allowedTypes** The type must be a component definition type.
- **ComponentParameter.typeInScope** The component definition reference must satisfy the scoping rules.

3.2.10 Equation (class)

An equation specifying the defining expression of an algebraic variable, or the derivative of a continuous variable.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *Equation*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

attr **derivative** [1] : *EBoolean*

Indicates whether this equation specifies the derivative of a continuous variable (*true*) or the defining expression of an algebraic variable (*false*).

cont **value** [1] : *Expression*

The defining value (expression) of the algebraic variable, or the derivative of the continuous variable.

Constraints:

- **Equation.varValueMatch** The type of the expression must match the type of the variable. That is, for algebraic variables, it must match the type of the algebraic variable declaration, and for derivatives of continuous variables, it must be a real type. For integer types of algebraic variables, the type of the value must be entirely contained in the type of the algebraic variable.
- **Equation.selfReference** The value of an equation may not, directly or indirectly, refer to the variable (that is, the algebraic variable, or the derivative) of the equation. That is, the algebraic variable or derivative may not be defined in terms of itself. This constraint is needed to satisfy the *AlgVariable.selfReference* and *ContVariable.selfReference* constraints.

ref **variable** [1] : *Declaration*

The continuous or algebraic variable that is given a value.

Constraints:

- **Equation.variableInScope** The continuous or algebraic variable must be declared in the same (sub-)scope as where the equation is located.
- **Equation.variableType** If *derivative* is *true*, then the variable that is referenced must be a continuous variable. If *derivative* is *false*, then the variable that is referenced must be an algebraic variable.

3.2.11 EventParameter (class)

An event parameter of a *ComponentDef* (Section 3.2.7).

Constraints:

- **EventParameter.flagChannelOnly** The ‘send’ (!), ‘receive’ (?), and ‘synchronization’ (~) flags may only be specified for channel parameters (event parameters with a data type, including *VoidType* (Section 3.5.19)).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Parameter* (Section 3.2.16)

⊆ *EventParameter*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **event** [1] : *Event*
The event declaration that represents the event parameter.

attr **recvFlag** [1] : *EBoolean*
Receive flag. If set, or if all flags unset, allows receiving use of the event passed via this parameter. If not set, and any of the other flags are set, receiving is not allowed.

attr **sendFlag** [1] : *EBoolean*

Send flag. If set, or if all flags unset, allows sending use of the event passed via this parameter. If not set, and any of the other flags are set, sending is not allowed.

attr **syncFlag** [1] : *EBoolean*

Synchronization flag. If set, or if all flags unset, allows synchronizing use of the event passed via this parameter. If not set, and any of the other flags are set, synchronizing is not allowed.

3.2.12 Group (class)

A group component. May contain other components, thus forming a component hierarchy. This hierarchy can be used to structure the CIF model.

All invariants at this level are combined using logical conjunction operators. The child component invariants are combined using logical conjunction operators as well. Assuming we have local invariants i_1, i_2, \dots, i_n , and we have child components c_1, c_2, \dots, c_m , with child component invariants j_1, j_2, \dots, j_m , then the invariant for the entire group becomes:

$$\bigwedge_{a=1}^n i_a \wedge \bigwedge_{b=1}^m j_b$$

In a similar way, the local initialization predicates are combined with the child component initialization predicates, and the local marker predicates are combined with the child component marker predicates.

Note that this uses mathematical conjunctions, and not the short-circuit conjunction binary operators. As such, there is no ordering between invariants, between initialization predicates, and between marker predicates.

Constraints:

- **Group.uniqueDecls** The names of all declarations, invariants, child components, and component definitions defined in a group must be unique within that group.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *Component* (Section 3.2.6)

⊢ *ComplexComponent* (Section 3.2.5)

⊢ *Group*

Direct derived classes: *Specification* (Section 3.2.17)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

attr **name** [1] : *CifIdentifier* (inherited from *Component*)

The name of the component.

Constraints:

- **Component.name (non-fatal)** Component names must not start with *e*-, *c*-, or *u*-, as those prefixes are reserved for events.

cont **declarations** [0..*] : *Declaration* (inherited from *ComplexComponent*)
The declarations of the component.

cont **equations** [0..*] : *Equation* (inherited from *ComplexComponent*)
The equations of the component.

cont **initials** [0..*] : *Expression* (inherited from *ComplexComponent*)
The initialization predicates of the component. A CIF model can only start in states that satisfy the initialization predicates.

If no predicates are given, the initialization predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between initialization predicates.

See the documentation of the derived classes for information on how to combine the initialization predicates with other levels.

Constraints:

- **ComplexComponent.initialTypes** The initialization predicates must have boolean types.

cont **invariants** [0..*] : *Invariant* (inherited from *ComplexComponent*)
The invariants of the component. A CIF model can only be in a state if that state satisfies the invariants.

If no invariants are given, the invariant is *true*. If multiple invariant are given, this feature represents the logical conjunction of those invariants. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between invariants.

See the documentation of the derived classes for information on how to combine the invariants with other levels.

cont **ioDecls** [0..*] : *IoDecl* (inherited from *ComplexComponent*)
The I/O declarations of the component.

Constraints:

- **ComplexComponent.maxOneSvgFile** Must not contain more than one *SvgFile* (Section 3.8.2) declaration.
- **ComplexComponent.maxOnePrintFile** Must not contain more than one *PrintFile* (Section 3.9.3) declaration.

cont **markeds** [0..*] : *Expression* (inherited from *ComplexComponent*)
The marker predicates of the component. Used as liveness property for, among others, supervisory controller synthesis.

If no predicates are given, the marker predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between marker predicates.

See the documentation of the derived classes for information on how to combine the marker predicates with other levels.

Constraints:

- **ComplexComponent.markedTypes** The marker predicates must have boolean types.

cont **components** [0..*] : *Component*

The sub-components of the group component.

cont **definitions** [0..*] : *ComponentDef*

The component definitions of the group component.

3.2.13 Invariant (class)

An invariant, with optional name and supervisory kind. Can either represent a state (exclusion) invariant, or a state/event exclusion invariant.

Constraints:

- **Invariant.unique (non-fatal)** Invariants must be unique.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Invariant*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **event** [0..1] : *Expression*

The event to exclude.

This feature contains an *Expression* (Section 3.6.21) instance, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **Invariant.eventOccurrence** The ‘event’ feature must be set for state/event exclusion invariants, and must not be set for state (exclusion) invariants.
- **Invariant.eventRef** The event reference expression, if set, must refer to an event.
- **Invariant.eventInScope** The event reference, if set, must satisfy the scoping rules.

attr **invKind** [1] : *InvKind*

The invariant kind of the invariant. Indicates whether the invariant is a state (exclusion) invariant, or a state/event exclusion invariant, and if applicable, in what form it was written.

attr **name** [0..1] : *CifIdentifier*

The name of the invariant.

Constraints:

- **Invariant.name (non-fatal)** Invariant names must not start with *e*_, *c*_, or *u*_, as those prefixes are reserved for events.

cont **predicate** [1] : *Expression*

The predicate of the invariant. For a state invariant, it is the predicate that must hold in the state. For state/event exclusion invariants, it is the predicate that is the necessary/sufficient condition for the event to be enabled/disabled.

Constraints:

- **Invariant.type** The invariant predicate must have a boolean type.

attr **supKind** [1] : *SupKind*

The supervisory kind of the invariant.

3.2.14 IoDecl (abstract class)

An I/O declaration. I/O declarations are used to couple the CIF model to external input/output, outside of the simulation behavior/semantics of the model.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *IoDecl*

Direct derived classes: *Print* (Section 3.9.2), *PrintFile* (Section 3.9.3), *SvgCopy* (Section 3.8.1), *SvgFile* (Section 3.8.2), *SvgIn* (Section 3.8.3), *SvgMove* (Section 3.8.8), *SvgOut* (Section 3.8.9)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

3.2.15 LocationParameter (class)

A location parameter of a *ComponentDef* (Section 3.2.7).

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *Parameter* (Section 3.2.16)

⊢ *LocationParameter*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **location** [1] : *Location*

The location declaration that represents the location parameter.

Constraints:

- **LocationParameter.nameOnly** The *location* must only have a name, and no initialization predicates, marker predicates, invariants, edges, or urgency, since it is a placeholder for a location, and not an actual location.

3.2.16 Parameter (abstract class)

A parameter of a *ComponentDef* (Section 3.2.7).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Parameter*

Direct derived classes: *AlgParameter* (Section 3.2.4), *ComponentParameter* (Section 3.2.9), *EventParameter* (Section 3.2.11), *LocationParameter* (Section 3.2.15)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

3.2.17 Specification (class)

A CIF specification. Also called a CIF model. The root object of every CIF model is an instance of this class, which is a special form of *Group* (Section 3.2.12).

Constraints:

- **Specification.root** Specifications are the roots of all CIF models. That is, specifications have no parent. Also, the root of any CIF model must be exactly one object, which is a specification.
- **Specification.name** The name of any specification must be "**specification**". Note that in the CIF textual syntax, the name of a specification cannot be specified.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Component* (Section 3.2.6)

⊆ *ComplexComponent* (Section 3.2.5)

⊆ *Group* (Section 3.2.12)

⊆ *Specification*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

attr **name** [1] : *CifIdentifier* (inherited from *Component*)
The name of the component.

Constraints:

- **Component.name (non-fatal)** Component names must not start with *e*-, *c*-, or *u*-, as those prefixes are reserved for events.

cont **declarations** [0..*] : *Declaration* (inherited from *ComplexComponent*)
The declarations of the component.

cont **equations** [0..*] : *Equation* (inherited from *ComplexComponent*)
The equations of the component.

cont **initials** [0..*] : *Expression* (inherited from *ComplexComponent*)
The initialization predicates of the component. A CIF model can only start in states that satisfy the initialization predicates.

If no predicates are given, the initialization predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between initialization predicates.

See the documentation of the derived classes for information on how to combine the initialization predicates with other levels.

Constraints:

- **ComplexComponent.initialTypes** The initialization predicates must have boolean types.

cont **invariants** [0..*] : *Invariant* (inherited from *ComplexComponent*)
The invariants of the component. A CIF model can only be in a state if that state satisfies the invariants.

If no invariants are given, the invariant is *true*. If multiple invariant are given, this feature represents the logical conjunction of those invariants. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between invariants.

See the documentation of the derived classes for information on how to combine the invariants with other levels.

cont **ioDecls** [0..*] : *IoDecl* (inherited from *ComplexComponent*)
The I/O declarations of the component.

Constraints:

- **ComplexComponent.maxOneSvgFile** Must not contain more than one *SvgFile* (Section 3.8.2) declaration.
- **ComplexComponent.maxOnePrintFile** Must not contain more than one *PrintFile* (Section 3.9.3) declaration.

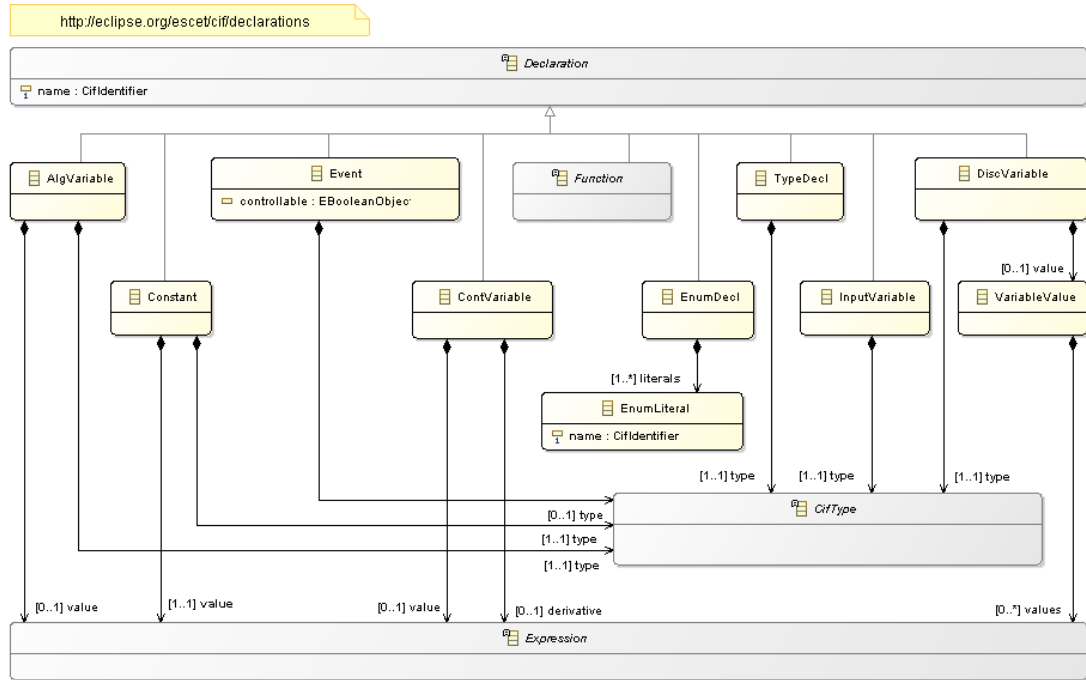


Figure 3.5: *declarations* package

cont **marked** [0..*] : *Expression* (inherited from *ComplexComponent*)

The marker predicates of the component. Used as liveness property for, among others, supervisory controller synthesis.

If no predicates are given, the marker predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between marker predicates.

See the documentation of the derived classes for information on how to combine the marker predicates with other levels.

Constraints:

- **ComplexComponent.markedTypes** The marker predicates must have boolean types.

cont **components** [0..*] : *Component* (inherited from *Group*)

The sub-components of the group component.

cont **definitions** [0..*] : *ComponentDef* (inherited from *Group*)

The component definitions of the group component.

3.3 Package declarations

Figure 3.5 shows the *declarations* package. This package contains declaration classes used to declare ‘small’ objects (compared to for instance component definitions). The *Declaration* (Section 3.3.4) class is the main class. All declarations have a name. Note that the *Function* (Section 3.7.6) class is further defined in the *functions* package.

Package URI <http://eclipse.org/escet/cif/declarations>

Namespace prefix declarations

Sub-packages none

3.3.1 AlgVariable (class)

An algebraic variable declaration. An algebraic variable is a shortcut for an expression. Everywhere the algebraic variable is used, its defining expression could be used instead. Note that references to other objects in the expression that defines the algebraic variable, are resolved in the context where the algebraic variable is defined, and not in the context where the algebraic variable is used.

A difference with constants is that constants must have constant values, while algebraic variables may depend on the values of for instance variables and locations. Another difference is in scoping. See also Section 3.1.

Note that in the CIF textual syntax, the range of an integer type of an algebraic variable, is optional. If it is not specified, and a range can be determined for the value of the algebraic variable, that range is used as the range of the type of the algebraic variable as well. This only applies to algebraic variables that have an integer type directly. It for instance does not apply to lists of integers.

Constraints:

- **AlgVariable.typeValueMatch** The *type* and *value* (if specified) of the algebraic variable must match. This constraint does not apply to algebraic parameters. For integer types, the range of the type of the value must be entirely contained in the range of the type of the algebraic variable.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
 - ⊆ *AnnotatedObject* (Section 3.10.1)
 - ⊆ *Declaration* (Section 3.3.4)
 - ⊆ *AlgVariable*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)

The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)

The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**-, **c**-, or **u**-, as those names are reserved for events.

cont **type** [1] : *CifType*

The type of the algebraic variable.

Constraints:

- **AlgVariable.allowedTypes** Component and component definition types are not allowed for algebraic variables.

cont **value** [0..1] : *Expression*

The value of the algebraic variable (the defining expression).

Constraints:

- **AlgVariable.selfReference** The value of an algebraic variable may not, directly or indirectly, refer to the algebraic variable itself. That is, an algebraic variable may not be defined in terms of itself. This includes the value of the algebraic variable (regardless of whether it is defined with the declaration or in one or more equations).
- **AlgVariable.uniqueValue** For each algebraic variable, the value must be uniquely defined. That is, either it is defined with the declaration itself (and not in any equation), or it is defined in a single equation in the same scope as where the variable is declared (and not in the declaration itself, not in multiple equations), or it is defined in an equation for each of the locations of the automaton that the variable is declared in (and not in the declaration itself, not in any equation in the automaton itself, not more than once per location).

For algebraic variables that are part of an algebraic parameter, the value must not be set, since then the arguments should provide the value. See also the *AlgParameter* (Section 3.2.4) class constraints.

3.3.2 Constant (class)

A constant declaration. A constant is a shortcut for a constant value. Everywhere the constant is used, its defining value could be used instead. Note that references to other objects in the value expression that defines the constant, are resolved in the context where the constant is defined, and not in the context where the constant is used.

A difference with algebraic variables is that constants must have constant values, while algebraic variable may depend on the values of for instance variables and locations. Another difference is in scoping. See also Section 3.1.

Note that in the CIF textual syntax, the range of an integer type of a constant, is optional. If it is not specified, and a range can be determined for the value of the constant, that range is used as the range of the type of the constant as well. This only applies to constants that have an integer type directly. It for instance does not apply to lists of integers.

Constraints:

- **Constant.typeValueMatch** The *type* and *value* of the constant must match. For integer types, the range of the type of the value must be entirely contained in the range of the type of the constant.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *AnnotatedObject* (Section 3.10.1)
 - ⊢ *Declaration* (Section 3.3.4)
 - ⊢ *Constant*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)

The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**-, **c**-, or **u**-, as those names are reserved for events.

cont **type** [1] : *CifType*

The type of the constant.

Constraints:

- **Constant.allowedTypes** Component and component definition types are not allowed for constants.

cont **value** [1] : *Expression*

The value of the constant (the defining expression).

Constraints:

- **Constant.selfReference** The value of a constant may not, directly or indirectly, refer to the constant itself. That is, a constant may not be defined in terms of itself.
- **Constant.constantValue** The value of a constant must be constant, and may thus not refer to objects that can change value, such as discrete and algebraic variables. Note that expressions may refer to locations ('is the location active'). Since they can change as well, location references are not allowed either.
Furthermore, user-defined functions are explicitly disallowed, as we can't statically determine whether they terminate.

3.3.3 ContVariable (class)

A continuous variable declaration. Continuous variables are implicitly real typed variables, which change value according to their derivative, as time passes.

Continuous variable declared in an *Automaton* (Section 3.4.3), may only be assigned in the automaton in which they are declared. If they are not declared in an automaton, they may not be assigned anywhere. They can however be accessed (read) elsewhere.

Constraints:

- **ContVariable.selfReference** The initial value of a continuous variable may not, directly or indirectly, refer to the continuous variable itself. That is, a continuous variable may not be defined in terms of itself. This includes the initial value of the continuous variable, but not its derivative (regardless of whether it is defined with the declaration or in one or more equations), which may be defined in terms of the continuous variable itself.

The derivative of a continuous variable may not, directly or indirectly, refer to the derivative of that continuous variable itself. That is, a derivative may not be defined in terms of itself (regardless of whether it is defined with the declaration or in one or more equations).

- **ContVariable.uniqueDerivative** For each continuous variable, the derivative must be uniquely defined. That is, either it is defined with the declaration itself (and not in any

equation), or it is defined in a single equation in the same scope as where the variable is declared (and not in the declaration itself, not in multiple equations), or it is defined in an equation for each of the locations of the automaton that the variable is declared in (and not in the declaration itself, not in any equation in the automaton itself, not more than once per location).

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *AnnotatedObject* (Section 3.10.1)
 - ⊢ *Declaration* (Section 3.3.4)
 - ⊢ *ContVariable*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)
The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e_**, **c_**, or **u_**, as those names are reserved for events.

cont **derivative** [0..1] : *Expression*
The derivative of the continuous variable.

Constraints:

- **ContVariable.derivativeType** If specified, the type of the derivative must be a real type.

cont **value** [0..1] : *Expression*
The initial value of the continuous variable. If not specified, defaults to value 0.0.

Constraints:

- **ContVariable.valueType** If specified, the type of the initial value must be a real type.

3.3.4 Declaration (abstract class)

Base class for ‘small’ (compared to for instance component definitions) CIF object declarations.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *AnnotatedObject* (Section 3.10.1)
- ⊆ *Declaration*

Direct derived classes: *AlgVariable* (Section 3.3.1), *Constant* (Section 3.3.2), *ContVariable* (Section 3.3.3), *DiscVariable* (Section 3.3.5), *EnumDecl* (Section 3.3.6), *Event* (Section 3.3.8), *Function* (Section 3.7.6), *InputVariable* (Section 3.3.9), *TypeDecl* (Section 3.3.10)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier*
The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**_, **c**_, or **u**_, as those names are reserved for events.

3.3.5 DiscVariable (class)

A discrete variable declaration.

Discrete variables declared in automata are always local to the *Automaton* (Section 3.4.3) in which they are declared, and can be assigned (modified) only in that automaton. They can however be accessed (read) elsewhere.

This class is also used for the parameters and local variables of internal user-defined functions.

Constraints:

- **DiscVariable.typeValueMatch** Each of the possible values of the discrete variable, if specified, must match the type of the discrete variable. If the type of the discrete variable has a range, the ranges of the types of the values must be entirely contained in the range of the type of the discrete variable.
- **DiscVariable.occurrence** Discrete variables may not be declared in groups (including specifications). That is, they must be declared in automata.
They may also be used in function parameters and as local variables in functions, but then they are usually not called discrete variables.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
 - ⊢ *AnnotatedObject* (Section 3.10.1)
 - ⊢ *Declaration* (Section 3.3.4)
 - ⊢ *DiscVariable*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)
The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**_, **c**_, or **u**_, as those names are reserved for events.

cont **type** [1] : *CifType*
The type of the discrete variable.

Constraints:

- **DiscVariable.allowedTypes** Component and component definition types are not allowed for discrete variables.

cont **value** [0..1] : *VariableValue*

The initial value of the discrete variable. If not specified, it has the default value for its type. The default values for each type are:

- Booleans: *false*,
- Integers (ranged): the value in the range that is closest to zero,
- Integers (rangeless): *0*,
- Reals: *0.0*,
- Strings: *""*,
- List (rangeless): empty list,
- List (ranged): list with as few elements as possible, with each element the default value of the element type,
- Set: empty set,
- Dictionary: empty dictionary,
- Tuple: tuple with for each field, the default value for the type of the field,
- Function: function that returns the default value for the return type of the function,
- Distribution: constant distribution with value *false* for boolean distributions, *0* for integer distributions, and *0.0* for real distributions,
- Enumerations: the first enumeration literal defined in the enumeration.

Some examples of default values for different integer ranges:

Range	Default value
[0 .. 5]	0
[2 .. 5]	2
[-5 .. -2]	-2
[-5 .. 5]	0
[-5 .. 0]	0

Some examples of default values for different list types:

List type	Default value
list int	[]
list[3] int	[0, 0, 0]
list[2..5] real	[0.0, 0.0]

Constraints:

- **DiscVariable.selfReference** The values of a discrete variable may not, directly or indirectly, refer to the discrete variable itself. That is, a discrete variable may not be defined in terms of itself.

3.3.6 EnumDecl (class)

An enumeration declaration. An enumeration is a strongly typed set of values, and is often clearer than arbitrary numbers.

Note that for equality purposes, two enumerations that define the same sequence of values (same number of values, with the same names, in the same order), are considered compatible types, and their enumeration literals (the values) can be compared for equality.

Note that for scoping purposes, the literals are defined at the same level as the enumeration declaration itself. This ensures that in the CIF textual syntax, we don't have to prefix enumeration literals with the name of the enumeration declaration.

Constraints:

- **EnumDecl.uniqueLiterals** The names of the enumeration literals of the enumeration declarations must be unique with respect to the other literals defined in that same enumeration, as well as all the declarations at the same level as the enumeration declaration that the literals are a part of.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
 - ⊆ *AnnotatedObject* (Section 3.10.1)
 - ⊆ *Declaration* (Section 3.3.4)
 - ⊆ *EnumDecl*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)
The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**_, **c**_, or **u**_, as those names are reserved for events.

cont **literals** [1..*] : *EnumLiteral*

The enumeration literals (values) that make up this enumeration.

3.3.7 EnumLiteral (class)

An enumeration literal (value) as part of an *EnumDecl* (Section 3.3.6).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *EnumLiteral*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

attr **name** [1] : *CifIdentifier*

The name of the enumeration literal.

Constraints:

- **EnumLiteral.name (non-fatal)** Enumeration literal names must not start with **e**_, **c**_, or **u**_, as those prefixes are reserved for events.

3.3.8 Event (class)

An event declaration.

Note that events that are not in the alphabet of any automaton, are globally disabled, since the alphabet of the specification is the union of the alphabets of the automata. In other words, if an event is not used on any edge, it is globally disabled. Event ‘tau’ is an exception, as it is never in the alphabet, since ‘tau’ does not synchronize. Channels are exceptions as well, as they require exactly one sender and one receiver, and if not further restricted by any synchronizing automata, they are pure channels.

Constraints:

- **Event.matchNameControllability (non-fatal)** If the name of the event starts with **c**_, the event must be defined to be controllable. If the name of the event starts with **u**_, the event must be defined to be uncontrollable. If the name of the event starts with **e**_, the event must not be defined to be controllable or uncontrollable. This is to ensure that the naming convention (which is also used for syntax highlighting), is not violated.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
 - ⊢ *AnnotatedObject* (Section 3.10.1)
 - ⊢ *Declaration* (Section 3.3.4)
 - ⊢ *Event*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)
The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with *e_*, *c_*, or *u_*, as those names are reserved for events.

attr **controllable** [0..1] : *EBooleanObject*
Indicates the controllability of the event. If *true*, the event is controllable. If *false*, the event is uncontrollable. If *null*, the event is not defined as controllable or uncontrollable.

cont **type** [0..1] : *CifType*
The type of the data communicated via this event, if applicable.

Constraints:

- **Event.allowedTypes** Component and component definition types are not allowed for events.

3.3.9 InputVariable (class)

An input variable declaration. An input variable is a variable for which the value is not defined by the CIF specification, but by an external source. It is allowed for the values of input variables to change at any time (discontinuously), depending on the external source. Input variables are read-only in the CIF specification.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *AnnotatedObject* (Section 3.10.1)
 - ⊢ *Declaration* (Section 3.3.4)
 - ⊢ *InputVariable*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)
The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**_, **c**_, or **u**_, as those names are reserved for events.

cont **type** [1] : *CifType*
The type of the input variable.

Constraints:

- **InputVariable.allowedTypes** Component and component definition types are not allowed for input variables.

3.3.10 TypeDecl (class)

A type declaration. Defines a reusable type. Mostly useful for types that have somewhat larger textual forms, which can then be referred to by name.

EObject

- ⊢ *PositionObject* (Section 3.11.2)

- ⊢ *AnnotatedObject* (Section 3.10.1)
 - ⊢ *Declaration* (Section 3.3.4)
 - ⊢ *TypeDecl*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)
The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**_, **c**_, or **u**_, as those names are reserved for events.

cont **type** [1] : *CifType*
The type of the type declaration. This type and the type declaration have compatible (equal) types.

Constraints:

- **TypeDecl.allowedTypes** Component and component definition types are not allowed.
- **TypeDecl.selfReference** The type of a type declaration may not, directly or indirectly, refer to the type declaration itself. That is, a type declaration may not be defined in terms of itself.

3.3.11 VariableValue (class)

Set of values that represents possible initial values of a *DiscVariable* (Section 3.3.5).

EObject

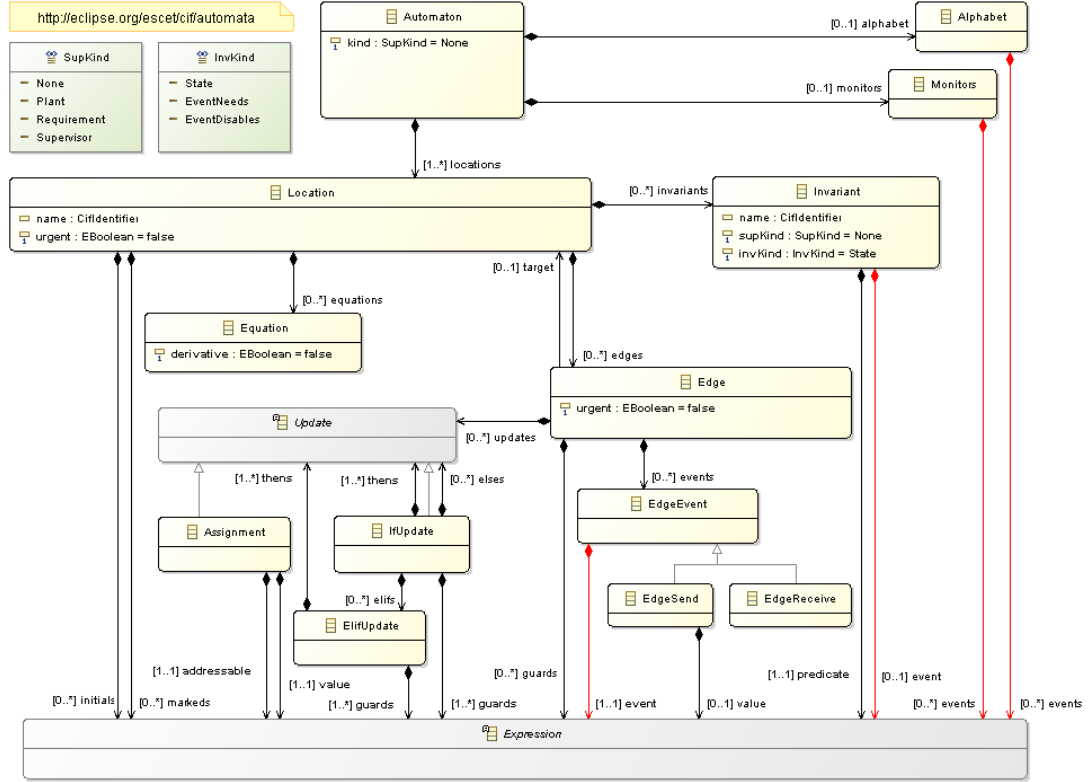


Figure 3.6: *automata* package

- └ *PositionObject* (Section 3.11.2)
 - └ *VariableValue*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **values** [0..*] : *Expression*

The possible initial values of the variable. If one or more values are specified, the initial value of the variable is one of those values. If no value is given, the variable can have any value in its domain.

3.4 Package automata

Figure 3.6 shows the *automata* package. This package contains classes related to automata. The *Automaton* (Section 3.4.3) class is the main class. Automata have at least one location. They may optionally also have an alphabet. Locations may have outgoing edges, and edges may have guards, updates, and events. This corresponds to most other automaton-based languages.

Package URI <http://eclipse.org/escet/cif/automata>

Namespace prefix automata

Sub-packages none

3.4.1 Alphabet (class)

An alphabet of an *Automaton* (Section 3.4.3).

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *Alphabet*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **events** [0..*] : *Expression*

The events that define the alphabet. May be empty to denote an empty alphabet. It is not possible to include the ‘tau’ event in alphabets.

This feature contains *Expression* (Section 3.6.21) instances, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **Alphabet.uniqueEvents (non-fatal)** Alphabets must contain unique events.
- **Alphabet.eventRefsOnly** The expressions must refer to events.
- **Alphabet.eventsInScope** The event references must satisfy the scoping rules.
- **Alphabet.eventParamSync** If an event reference refers to an event parameter, the event parameter should allow synchronization use. That is, the event parameter should not specify any flags (allow all usages) or should explicitly specify the synchronization (\sim) flag to allow synchronization.

3.4.2 Assignment (class)

An assignment update.

Constraints:

- **Assignment.types** The type of the assigned value must match the type of the addressable that is assigned. If the variable being assigned has a ranged type, then the range of the type of the value must have overlap with the range of the type of the variable. It is considered a run-time error if the evaluated value expression results in a value that is outside of the range of the assigned variable.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *Update* (Section 3.4.12)
- ⊆ *Assignment*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **addressable** [1] : *Expression*

The addressable (a variable, a part of a variable, multiple variables, parts of multiple variables, etc) to which to assign a value.

It is allowed to create a new key/value pair in a dictionary, if the key of the last projection does not exist. It is not allowed to create a new key/value pair, if the key of one of the other projections does not exist.

The indices of the projections of the addressables are evaluated in the ‘old’ state.

Constraints:

- **Assignment.addressableSyntax** Addressables on edges may be discrete and continuous variable references (non-wrapped) with projections, and addressables in CIF/SVG input mappings may be input variable references (non-wrapped) with projections. The variable references may optionally be wrapped in tuples (possibly multiple times). Projected string typed variables are not allowed.
- **Assignment.variablesInScope** The variables that are assigned on edges must be discrete or continuous variables declared in the automaton that contains the source location of the edge of this assignment update. Discrete and continuous variables may be mixed in a single assignment. The variables that are assigned in CIF/SVG input mappings must be input variables.

cont **value** [1] : *Expression*

The value to assign to the variable(s).

This expression is evaluated in the ‘old’ state.

3.4.3 Automaton (class)

An automaton component. This class is the basic leaf component of all component hierarchies.

All invariants at this level are combined using logical conjunction operators. The location invariants are combined with the location itself, using logical implication operators. Assuming we have local invariants i_1, i_2, \dots, i_n , and we have locations l_1, l_2, \dots, l_m , with location invariants j_1, j_2, \dots, j_m , then the invariant for the entire automaton becomes:

$$\bigwedge_{a=1}^n i_a \wedge \bigwedge_{b=1}^m l_b \Rightarrow j_b$$

In a similar way, the local initialization predicates are combined with the location initialization predicates, and the local marker predicates are combined with the location marker predicates.

Note that this uses mathematical conjunctions, and not the short-circuit conjunction binary operators. As such, there is no ordering between invariants, between initialization predicates, and between marker predicates.

Constraints:

- **Automaton.uniqueDecls** The names of all declarations, invariants and locations defined in an automaton must be unique within that automaton.
- **Automaton.validAlphabet** If an alphabet is specified, it must contain at least the events that occur on the edges of this automaton (excluding communication usage by means of sends and receives). Note that the ‘tau’ event is never part of alphabets.
- **Automaton.noFuncDecl** Functions may not be defined in automata.
- **Automaton.uniqueUsagePerEvent** Each automaton must use a certain event in at most one way: synchronization, sending, or receiving. The events that are used to synchronize are determined by the alphabet. The send/receive uses are found via the edges.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *Component* (Section 3.2.6)
 - ⊢ *ComplexComponent* (Section 3.2.5)
 - ⊢ *Automaton*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

attr **name** [1] : *CifIdentifier* (inherited from *Component*)
The name of the component.

Constraints:

- **Component.name (non-fatal)** Component names must not start with *e*-, *c*-, or *u*-, as those prefixes are reserved for events.

cont **declarations** [0..*] : *Declaration* (inherited from *ComplexComponent*)
The declarations of the component.

cont **equations** [0..*] : *Equation* (inherited from *ComplexComponent*)
The equations of the component.

cont **initials** [0..*] : *Expression* (inherited from *ComplexComponent*)
The initialization predicates of the component. A CIF model can only start in states that satisfy the initialization predicates.

If no predicates are given, the initialization predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this

represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between initialization predicates.

See the documentation of the derived classes for information on how to combine the initialization predicates with other levels.

Constraints:

- **ComplexComponent.initialTypes** The initialization predicates must have boolean types.

cont **invariants** [0..*] : *Invariant* (inherited from *ComplexComponent*)

The invariants of the component. A CIF model can only be in a state if that state satisfies the invariants.

If no invariants are given, the invariant is *true*. If multiple invariant are given, this feature represents the logical conjunction of those invariants. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between invariants.

See the documentation of the derived classes for information on how to combine the invariants with other levels.

cont **ioDecls** [0..*] : *IoDecl* (inherited from *ComplexComponent*)

The I/O declarations of the component.

Constraints:

- **ComplexComponent.maxOneSvgFile** Must not contain more than one *SvgFile* (Section 3.8.2) declaration.
- **ComplexComponent.maxOnePrintFile** Must not contain more than one *PrintFile* (Section 3.9.3) declaration.

cont **marked** [0..*] : *Expression* (inherited from *ComplexComponent*)

The marker predicates of the component. Used as liveness property for, among others, supervisory controller synthesis.

If no predicates are given, the marker predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between marker predicates.

See the documentation of the derived classes for information on how to combine the marker predicates with other levels.

Constraints:

- **ComplexComponent.markedTypes** The marker predicates must have boolean types.

cont **alphabet** [0..1] : *Alphabet*

The alphabet of the automaton. If not specified, it defaults to the events used on the edges of the automaton (excluding communication usage by means of sends and receives). Note that the ‘tau’ event is never part of alphabets.

attr **kind** [1] : *SupKind*

The supervisory kind of the automaton.

cont **locations** [1..*] : *Location*

The locations of the automaton.

cont **monitors** [0..1] : *Monitors*

The monitor events of the automaton. For details on monitors, see the *Monitors* (Section 3.4.11) class.

If not specified, the automaton does not monitor any events.

3.4.4 Edge (class)

An outgoing edge of a *Location* (Section 3.4.10) of an *Automaton* (Section 3.4.3).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Edge*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **events** [0..*] : *EdgeEvent*

The events of the edge. Specifying multiple events is equivalent to duplicating the edge, with each edge containing one of the events of the original edge. Specifying no events is equivalent to specifying a single ‘tau’ event.

This feature contains *Expression* (Section 3.6.21) instances, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **Edge.uniqueEvents (non-fatal)** Edges must contain unique events.
- **Edge.oneReceiveAllReceive** If one of the events performs a receive, the other events on that same edge must also perform receives, and the data types of their events must be equal. This is to ensure that the received value reference expression (if used), is available, and has a consistent type.

We don’t allow unequal but compatible types, as that would lead to a union of those types for the received value reference expression, and that would lead to a difference in type for multiple events on a single edge, and multiple edges with one event each.

cont **guards** [0..*] : *Expression*

The guard predicates of the edge. An edge is only enabled if the guard evaluates to true in the current state.

If no predicates are given, the guard predicate is *true*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between the guards of an edge.

Constraints:

- **Edge.guardTypes** The guard predicates must have boolean types.

ref **target** [0..1] : *Location*

The target location (within this automaton) of the edge. If not set, the edge is a self-loop, which means that the target location of the edge is equal to the source location of the edge.

Constraints:

- **Edge.targetInScope** The target location of the edge, if specified, must be declared in the same automaton as the source location of the edge. In particular, location parameters of component definitions are not allowed.

cont **updates** [0..*] : *Update*

The updates to execute if the edge is ‘executed’.

Constraints:

- **Edge.uniqueVariables** The parts of the variables that are assigned must be unique. That is, it must never be possible to assign the same part of the same variable twice. Since we don’t include the guards of conditional updates in the analysis, this may lead to false positives. We believe that in such cases, the update can easily be rewritten, and thus the false positives will not be a problem in practice.

We include the projections in the analysis. For tuple projections, the index can always be statically evaluated, as the type checker needs the index to get the type of the field. Tuple field references are normalized to 0-based field indices. For lists, indices are evaluated if this is possible statically. Negative array indices are normalized. For non-array lists, negative indices can not be normalized statically. For dictionaries, the keys are statically evaluated, if possible. Since not all expressions can be statically evaluated, and also normalization is not always possible statically, overlap of parts of variables can not always be properly detected, potentially leading to false positives. We opt for false positives rather than runtime checks, to avoid having to handle such cases in many tools and transformations.

attr **urgent** [1] : *EBoolean*

Whether the edge is urgent. If an edge is urgent, and the guard is enabled, then time may not progress.

Constraints:

- **Edge.urgWhenLocUrg (non-fatal)** An edge should not be urgent when the source location of the edge is also urgent, as that adds no additional constraint or behavior.

3.4.5 EdgeEvent (class)

An event of an edge. Optionally with communication, see derived classes.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *EdgeEvent*

Direct derived classes: *EdgeReceive* (Section 3.4.6), *EdgeSend* (Section 3.4.7)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **event** [1] : *Expression*
The event reference of the edge event.

This feature contains an *Expression* (Section 3.6.21) instance, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **EdgeEvent.eventRefsOnly** The expression must refer to an event.
- **EdgeEvent.eventsInScope** The event references must satisfy the scoping rules.
- **EdgeEvent.eventParamUse** If an event reference refers to an event parameter, the event parameter should allow the use of the event as it is used on this edge. That is, the event parameter should not specify any flags (allow all usages) or should explicitly specify the flag that allows the usage on this edge.

3.4.6 EdgeReceive (class)

A receive event of an edge.

Constraints:

- **EdgeReceive.commEvent** The *event* must have a data type.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *EdgeEvent* (Section 3.4.5)

⊢ *EdgeReceive*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **event** [1] : *Expression* (inherited from *EdgeEvent*)
The event reference of the edge event.

This feature contains an *Expression* (Section 3.6.21) instance, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **EdgeEvent.eventRefsOnly** The expression must refer to an event.
- **EdgeEvent.eventsInScope** The event references must satisfy the scoping rules.
- **EdgeEvent.eventParamUse** If an event reference refers to an event parameter, the event parameter should allow the use of the event as it is used on this edge. That is, the event parameter should not specify any flags (allow all usages) or should explicitly specify the flag that allows the usage on this edge.

3.4.7 EdgeSend (class)

A send event of an edge.

Constraints:

- **EdgeSend.commEvent** The *event* must have a data type.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *EdgeEvent* (Section 3.4.5)

⊢ *EdgeSend*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **event** [1] : *Expression* (inherited from *EdgeEvent*)

The event reference of the edge event.

This feature contains an *Expression* (Section 3.6.21) instance, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **EdgeEvent.eventRefsOnly** The expression must refer to an event.
- **EdgeEvent.eventsInScope** The event references must satisfy the scoping rules.
- **EdgeEvent.eventParamUse** If an event reference refers to an event parameter, the event parameter should allow the use of the event as it is used on this edge. That is, the event parameter should not specify any flags (allow all usages) or should explicitly specify the flag that allows the usage on this edge.

cont **value** [0..1] : *Expression*

The value to send as part of the communication. This expression is evaluated in the ‘old’ state, before any updates are processed.

Constraints:

- **EdgeSend.allowedValue** If the *event* has a non-void data type, then the *value* is required. If the data type is *VoidType* (Section 3.5.19), then a *value* is not allowed.
- **EdgeEvent.valueType** The type of the value must match the type of the event (first contained in the second).

3.4.8 ElifUpdate (class)

An ‘elif’ (‘else-if’) alternative of an *IfUpdate* (Section 3.4.9).

EObject

\sqsubset *PositionObject* (Section 3.11.2)
 \sqsubset *ElifUpdate*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

cont **guards** [1..*] : *Expression*
 The guard predicates for this alternative.

If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between the guards.

Constraints:

- **ElifUpdate.guardTypes** The guard predicates must have boolean types.

cont **thens** [1..*] : *Update*
 The update to execute for the *ElifUpdate* (Section 3.4.8) if the ‘guards’ evaluate to *true*.

3.4.9 IfUpdate (class)

A conditional update.

EObject
 \sqsubset *PositionObject* (Section 3.11.2)
 \sqsubset *Update* (Section 3.4.12)
 \sqsubset *IfUpdate*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

cont **elifs** [0..*] : *ElifUpdate*
 The ‘else-if’ (‘elif’) alternatives. Processed in order, if the ‘guards’ evaluate to *false*.

cont **elses** [0..*] : *Update*
 The ‘else’ updates. Executed if no other alternative has a *true* guard.

cont **guards** [1..*] : *Expression*
 The guard predicates for the ‘then’ updates.

If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between the guards.

Constraints:

- **IfUpdate.guardTypes** The guard predicates must have boolean types.

cont **thens** [1..*] : *Update*

The ‘then’ updates. Executed if the ‘guards’ evaluate to *true*.

3.4.10 Location (class)

A location of an *Automaton* (Section 3.4.3).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *AnnotatedObject* (Section 3.10.1)

⊆ *Location*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)

The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

cont **edges** [0..*] : *Edge*

The outgoing edges of the location.

cont **equations** [0..*] : *Equation*

The equations of the location.

cont **initials** [0..*] : *Expression*

The initialization predicates of the location. A CIF model can only start in states that satisfy the initialization predicates.

If no predicates are given, the initialization predicate is *false*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between initialization predicates.

Constraints:

- **Location.initialTypes** The initialization predicates must have boolean types.

cont **invariants** [0..*] : *Invariant*

The invariants of the location. A CIF model can only be in a state if that state satisfies the invariants.

If no invariants are given, the invariant is *true*. If multiple invariants are given, this feature represents the logical conjunction of those invariants. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between invariants.

cont **marked**s [0..*] : *Expression*

The marker predicates of the location. Used as liveness property for, among others, supervisory controller synthesis.

If no predicates are given, the marker predicate is *false*. If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between marker predicates.

Constraints:

- **Location.markedTypes** The marker predicates must have boolean types.

attr **name** [0..1] : *CifIdentifier*

The name of the location. If a location is the only location in an automaton, the name may be omitted, but then the location may not be referenced.

Constraints:

- **Location.nameless** The name of a location must be set if one of the following conditions holds: (1) the location is not the only location of the automaton; (2) the location is referenced; (3) the location is a location parameter. In other words, if it is not a location parameter, it is the only location of the automaton, and it is never referenced, the name may be omitted.

As for (2), it is not possible to refer to a nameless location in the CIF textual syntax. However, it is possible to do that in the metamodel, and as such the situation can result from for instance a model transformation.

- **Location.name (non-fatal)** Location names must not start with *e*_, *c*_, or *u*_, as those prefixes are reserved for events.

attr **urgent** [1] : *EBoolean*

Whether the location is urgent. If a location is urgent, then time may not progress in that location.

3.4.11 Monitors (class)

The monitor events of an *Automaton* (Section 3.4.3).

Automata that monitor events are used to monitor (track, observe) the behavior (or progress) of those monitor events, as initiated by other components. Monitoring is often used for verification components, supervisors, etc.

Informally, if an automaton monitors an event, the event is never blocked by that automaton (guard-wise).

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *Monitors*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

cont **events** [0..*] : *Expression*
 The set of monitor events.

If no events are specified for this feature, the automaton monitors all events in its alphabet. This differs from an automaton not having a *Monitors* (Section 3.4.11) instance at all, as that means the automaton does not monitor any events at all.

This feature contains *Expression* (Section 3.6.21) instances, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **Automaton.monitorsUniqueEvents (non-fatal)** The set of monitor events must contain unique events.
- **Automaton.monitorsEventRefsOnly** The expressions must refer to events. Event ‘tau’ is not allowed, as that event is never part of the alphabet, as it does not synchronize with other automata.
- **Automaton.monitorsEventsInScope** The event references must satisfy the scoping rules.
- **Automaton.monitorsSubsetAlphabet** The set of monitor events must be a subset of the alphabet.

3.4.12 Update (abstract class)

An update of an *Edge* (Section 3.4.4) or *SvgIn* (Section 3.8.3).

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *Update*

Direct derived classes: *Assignment* (Section 3.4.2), *IfUpdate* (Section 3.4.9)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

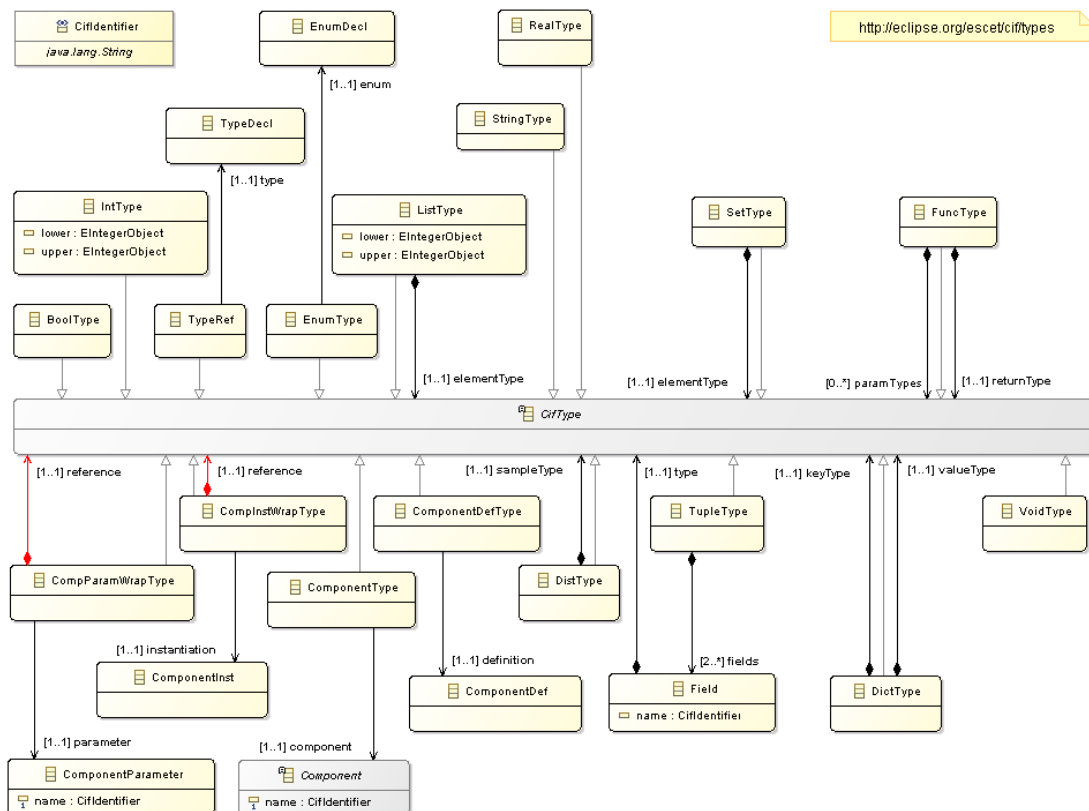


Figure 3.7: *types* package

3.5 Package types

Figure 3.7 shows the *types* package. This package contains type classes, and is pretty standard compared to other languages. The *CifType* (Section 3.5.2) class is the main class. Of particular interest may be the *CompParamWrapType* (Section 3.5.4) and *CompInstWrapType* (Section 3.5.3) classes. They are used to keep track of references via component parameters and component instantiations. See also Section 3.1.

Package URI <http://eclipse.org/escet/cif/types>

Namespace prefix *types*

Sub-packages none

3.5.1 BoolType (class)

A boolean type. Represents boolean values.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
 - ⊢ *CifType* (Section 3.5.2)
 - ⊢ *BoolType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

3.5.2 CifType (abstract class)

Base class for all CIF types.

EObject
 ⊢ *PositionObject* (Section 3.11.2)
 ⊢ *CifType*

Direct derived classes: *BoolType* (Section 3.5.1), *CompInstWrapType* (Section 3.5.3), *CompParamWrapType* (Section 3.5.4), *ComponentDefType* (Section 3.5.5), *ComponentType* (Section 3.5.6), *DictType* (Section 3.5.7), *DistType* (Section 3.5.8), *EnumType* (Section 3.5.9), *FuncType* (Section 3.5.11), *IntType* (Section 3.5.12), *ListType* (Section 3.5.13), *RealType* (Section 3.5.14), *SetType* (Section 3.5.15), *StringType* (Section 3.5.16), *TupleType* (Section 3.5.17), *TypeRef* (Section 3.5.18), *VoidType* (Section 3.5.19)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

3.5.3 CompInstWrapType (class)

Component instantiation reference wrapping type. Allows keeping track of references via component instantiations. Similar to the *CompInstWrapExpression* (Section 3.6.9) class, but for type references instead of expression references. Similar to the *CompParamWrapType* (Section 3.5.4) class, but for references via component instantiations instead of via component parameters. See also Section 3.1.

EObject
 ⊢ *PositionObject* (Section 3.11.2)
 ⊢ *CifType* (Section 3.5.2)
 ⊢ *CompInstWrapType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

ref **instantiation** [1] : *ComponentInst*

The component instantiation via which the object is referenced.

Constraints:

- **CompInstWrapType.noCompDefBody** Components instantiations that are bodies of component definitions may not be referenced here, as the body *is* the component definition.
- **CompInstWrapType.instantiationInScope** The component instantiation reference must satisfy the scoping rules.

cont **reference** [1] : *CifType*

The object that is referenced via a component instantiation.

This feature contains *CifType* (Section 3.5.2) instances, to allow for wrapping types to be used.

Constraints:

- **CompInstWrapType.reference** The *reference* type must be a reference.

3.5.4 CompParamWrapType (class)

Component parameter reference wrapping type. Allows keeping track of references via component parameters. Similar to the *CompParamWrapExpression* (Section 3.6.11) class, but for type references instead of expression references. Similar to the *CompInstWrapType* (Section 3.5.3) class, but for references via component parameters instead of via component instantiations. See also Section 3.1.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *CifType* (Section 3.5.2)

⊆ *CompParamWrapType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

ref **parameter** [1] : *ComponentParameter*

The component parameter via which the object is referenced.

Constraints:

- **CompParamWrapType.parameterInScope** The component parameter reference must satisfy the scoping rules.

cont **reference** [1] : *CifType*

The object that is referenced via a component parameter.

This feature contains *CifType* (Section 3.5.2) instances, to allow for wrapping types to be used.

Constraints:

- **CompParamWrapType.reference** The *reference* type must be a reference.

3.5.5 ComponentDefType (class)

A component definition type.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *CifType* (Section 3.5.2)
- ⊆ *ComponentDefType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

ref **definition** [1] : *ComponentDef*
The component definition that is used as a type.

Constraints:

- **ComponentDefType.definitionInScope** The component definition reference must satisfy the scoping rules.

3.5.6 ComponentType (class)

A component type.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *CifType* (Section 3.5.2)
- ⊆ *ComponentType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

ref **component** [1] : *Component*
The component that is used as a type.

Constraints:

- **ComponentType.noCompDefBody** Components that are bodies of component definitions may not be referenced here, as the body *is* the component definition. In such cases a *ComponentDefType* (Section 3.5.5) should be used instead.

- **ComponentType.noSpec** Specifications, which are technically components, may not be referenced here, as they serve as outer grouping only. Note that in the CIF textual syntax, they can't be referenced either, as they don't have a name.
- **ComponentType.componentInScope** The component reference must satisfy the scoping rules.

3.5.7 DictType (class)

A dictionary type. Represents collections of key/value pairs, where keys are unique within a single dictionary.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *CifType* (Section 3.5.2)
- ⊆ *DictType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **keyType** [1] : *CifType*
The type of the keys of the dictionary.
Constraints:

- **DictType.allowedKeyTypes** Component and component definition types, as well as function and distribution types, are not allowed for dictionary keys.

cont **valueType** [1] : *CifType*
The type of the values of the dictionary.
Constraints:

- **DictType.allowedValueTypes** Component and component definition types are not allowed for dictionary values.

3.5.8 DistType (class)

A distribution type. Represents stochastic distributions that produce values (samples) according to a certain probability function.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *CifType* (Section 3.5.2)
- ⊆ *DistType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **sampleType** [1] : *CifType*
The type of the samples produces by the distribution.

Constraints:

- **DistType.allowedSampleTypes** Only booleans, integers (without range), and reals are allowed as sample types.

3.5.9 EnumType (class)

An enumeration type. Represents the enumeration literals of a certain enumeration.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *CifType* (Section 3.5.2)

⊆ *EnumType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

ref **enum** [1] : *EnumDecl*

The enumeration of which the enumeration literals make up this type.

Constraints:

- **EnumRef.enumInScope** The enumeration reference must satisfy the scoping rules.

3.5.10 Field (class)

A field of a tuple type.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Field*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

attr **name** [0..1] : *CifIdentifier*

The name of the field. Note that the name is mandatory in the CIF textual syntax, but is optional to allow for anonymous tuple fields that result from standard library functions, etc.

cont **type** [1] : *CifType*
The type of the field.

Constraints:

- **Field.allowedTypes** Component and component definition types are not allowed for tuple fields.

3.5.11 FuncType (class)

A function type. Represents functions.

EObject

⊆ *PositionObject* (Section 3.11.2)
⊆ *CifType* (Section 3.5.2)
⊆ *FuncType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **paramTypes** [0..*] : *CifType*
The types of the parameters of the function.
Constraints:

- **FuncType.allowedParamTypes** Component and component definition types are not allowed for function parameters.

cont **returnType** [1] : *CifType*
The type of the result of the function.
Constraints:

- **FuncType.allowedReturnTypes** Component and component definition types are not allowed for function return values.

3.5.12 IntType (class)

An integer type. Represents a subset of all integer values.

The range of integer types is optional. If not specified, the range is $[-2^{31} .. 2^{31} - 1]$. See also the *IntExpression* (Section 3.6.27) class.

Constraints:

- **IntType.neitherOrBoth** Either the *lower* bound and the *upper* bound are specified, or neither of them are.

- **IntType.validRange** If the bounds are specified, the *lower* bound value must be less than or equal to the *upper* bound value, to make sure we have a valid (non-empty) range.

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *CifType* (Section 3.5.2)
 ⊆ *IntType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

attr **lower** [0..1] : *EIntegerObject*
 The lower bound (inclusive) of the values that make up this type. If it is not specified, it is implicitly -2^{31} .

attr **upper** [0..1] : *EIntegerObject*
 The upper bound (inclusive) of the values that make up this type. If it is not specified, it is implicitly $2^{31} - 1$.

3.5.13 ListType (class)

A list type. Represents ordered collections of elements, possibly with duplicates.

The range of list types is optional. If not specified, the range is $[0 .. 2^{31} - 1]$. This follows from the use of integers for the ranges. See also the *IntType* (Section 3.5.12) and *IntExpression* (Section 3.6.27) classes.

Constraints:

- **ListType.neitherOrBoth** Either the *lower* bound and the *upper* bound are specified, or neither of them are.
- **ListType.nonNegativeRangeBounds** If the bounds are specified, they must be non-negative.
- **ListType.validRange** If the bounds are specified, the *lower* bound value must be less than or equal to the *upper* bound value, to make sure we have a valid (non-empty) range.

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *CifType* (Section 3.5.2)
 ⊆ *ListType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **elementType** [1] : *CifType*
The type of the elements of the list.

Constraints:

- **ListType.allowedTypes** Component and component definition types are not allowed for list elements.

attr **lower** [0..1] : *EIntegerObject*
The lower bound (inclusive) of the allowed sizes of the lists that are values of this type. If it is not specified, it is implicitly 0.

attr **upper** [0..1] : *EIntegerObject*
The upper bound (inclusive) of the allowed sizes of the lists that are values of this type. If it is not specified, it is implicitly $2^{31} - 1$.

3.5.14 RealType (class)

A real type. Represents a subset of all real values.

Real values are implemented using Java's `double` data type, which is conceptually associated with double-precision 64-bit format IEEE 754 values as specified in IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York). The IEEE 754 standard includes not only positive and negative numbers that consist of a sign and magnitude, but also positive and negative zeros, positive and negative infinities, and a special Not-a-Number (NaN) value. A NaN value is used to represent the result of certain invalid operations such as dividing zero by zero. We explicitly exclude positive and negative infinities, as well as NaN values, and any operation that results in them is considered a run-time evaluation error. Negative zero values are automatically converted to positive zero values. See also the *RealExpression* (Section 3.6.31) class.

EObject

⊆ *PositionObject* (Section 3.11.2)
⊆ *CifType* (Section 3.5.2)
⊆ *RealType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

3.5.15 SetType (class)

A set type. Represents unordered collections of elements, without duplicates.

EObject

⊢ *PositionObject* (Section 3.11.2)
⊢ *CifType* (Section 3.5.2)
⊢ *SetType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **elementType** [1] : *CifType*
The type of the elements of the set.

Constraints:

- **SetType.allowedTypes** Component and component definition types, as well as function and distribution types, are not allowed for set elements.

3.5.16 StringType (class)

A string type. Represents string values, in other words, it represents texts.

EObject

⊢ *PositionObject* (Section 3.11.2)
⊢ *CifType* (Section 3.5.2)
⊢ *StringType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

3.5.17 TupleType (class)

A tuple type. Represents ordered collections of elements, where the number of elements is fixed, and each element can have a different type.

EObject

⊢ *PositionObject* (Section 3.11.2)
⊢ *CifType* (Section 3.5.2)
⊢ *TupleType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **fields** [2..*] : *Field*
The fields of the tuple.

Constraints:

- **TupleType.uniqueFieldNames** The names of the fields of the tuple type must be unique with respect to the other fields defined in that same tuple type.

3.5.18 TypeRef (class)

A type reference. Represents the same values as the values of the referred type declaration.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *CifType* (Section 3.5.2)

⊆ *TypeRef*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

ref **type** [1] : *TypeDecl*
The *TypeDecl* (Section 3.3.10) that this type reference refers to.

Constraints:

- **TypeRef.typeInScope** The type declaration reference must satisfy the scoping rules.

3.5.19 VoidType (class)

A void type. Used to represent the type of a channel over which no data is communicated.

Constraints:

- **VoidType.occurrence** The ‘void’ type may only be used as type of events (and event parameters). It may not be used in other types (such as list types), in type declarations, etc.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *CifType* (Section 3.5.2)

⊆ *VoidType*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

3.6 Package expressions

Figure 3.8 shows the *expressions* package. This package contains expression classes, and is pretty standard compared to other languages.

The *Expression* (Section 3.6.21) class is the main class. Note that all expressions have a mandatory type. The derived classes above the *Expression* (Section 3.6.21) class represent literals, operators, collections, etc.

The derived classes below *Expression* (Section 3.6.21) all deal with references to objects in the CIF language. Of particular interest may be the *CompParamWrapExpression* (Section 3.6.11) and *CompInstWrapExpression* (Section 3.6.9) classes. They are used to keep track of references via component parameters and component instantiations. See also Section 3.1.

Package URI <http://eclipse.org/escet/cif/expressions>

Namespace prefix expressions

Sub-packages none

3.6.1 BinaryOperator (enumeration)

Binary operators for the *BinaryExpression* (Section 3.6.6). Type constraints are listed in the *BinaryExpression* (Section 3.6.6) class documentation.

literal **Addition**

Integer addition binary operator (+), as well as list concatenation, string concatenation, and dictionary update.

literal **BiConditional**

Logical biconditional (or ‘if and only if’) binary operator (\Leftrightarrow).

literal **Conjunction**

Logical conjunction binary operator (\wedge), as well as set intersection (\cap).

Note that even though expressions are side effect free, evaluation may still fail. Implementations must guarantee short circuit evaluation of this operator. Note that when manipulating expressions, the operands may only be switched if the resulting expression evaluates to the same value as the original expression did, when using short circuit evaluation semantics for both the original and resulting expression.

literal **Disjunction** (default)

Logical disjunction binary operator (\vee), as well as set union (\cup).

Note that even though expressions are side effect free, evaluation may still fail. Implementations must guarantee short circuit evaluation of this operator. Note that when manipulating expressions, the operands may only be switched if the resulting expression evaluates to the

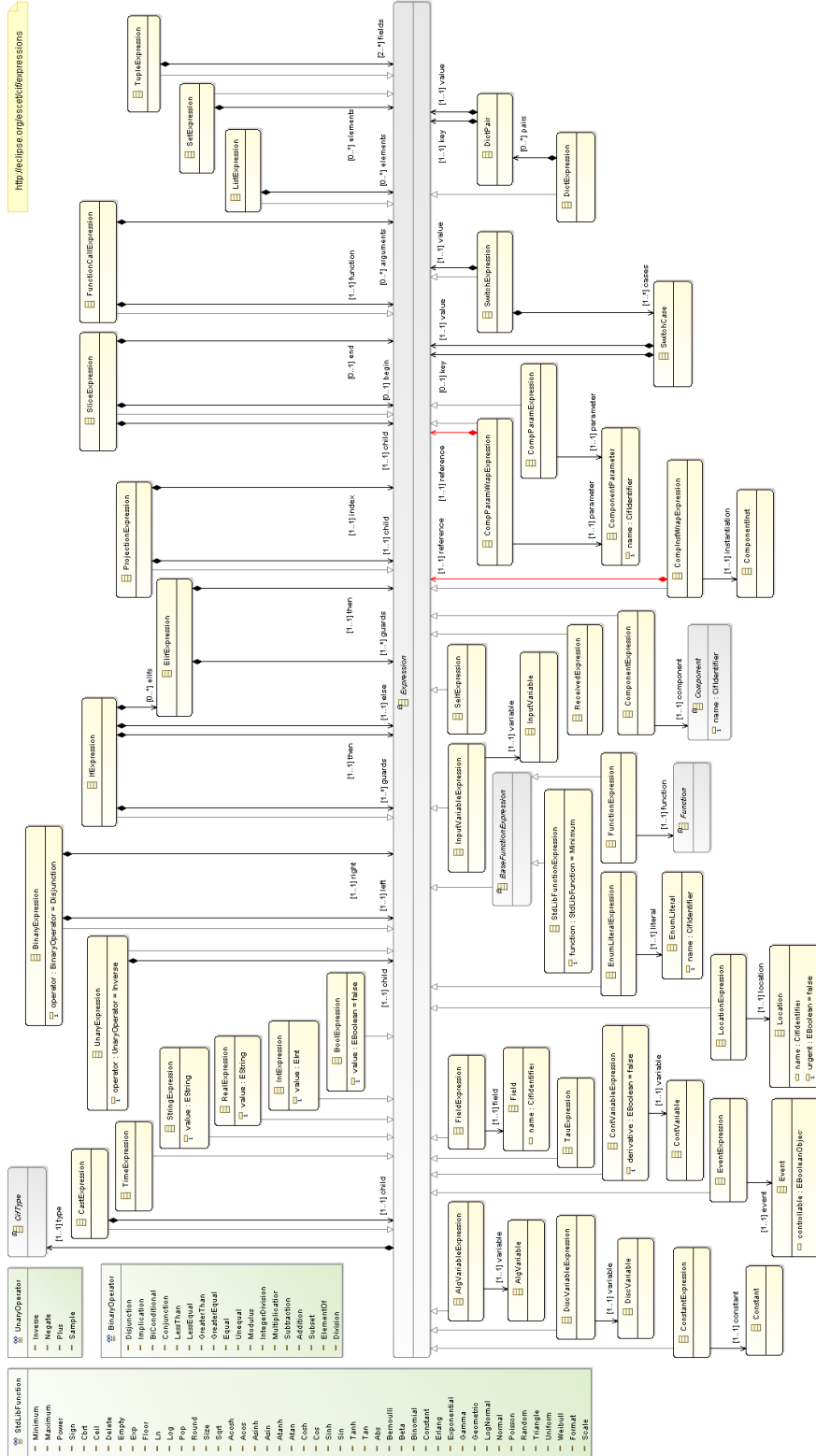


Figure 3.8: *expressions* package

same value as the original expression did, when using short circuit evaluation semantics for both the original and resulting expression.

literal **Division**

Real division binary operator (/).

Division by zero is considered a run-time error.

literal **ElementOf**

Element-of binary operator (\in).

literal **Equal**

Equality binary operator ($=$).

literal **GreaterEqual**

Greater than or equal comparison binary operator (\geq).

literal **GreaterThan**

Greater than comparison binary operator ($>$).

literal **Implication**

Logical implication (or material condition, or material implication) binary operator (\Rightarrow).

Note that even though expressions are side effect free, evaluation may still fail. Implementations must guarantee short circuit evaluation of this operator. Note that when manipulating expressions, the operands may only be switched if the resulting expression evaluates to the same value as the original expression did, when using short circuit evaluation semantics for both the original and resulting expression.

literal **IntegerDivision**

Integer division binary operator (div). Note that this operator has the truncated division, or round towards zero, semantics.

a	b	$a \text{ div } b$	$a \text{ mod } b$
7	4	1	3
7	-4	-1	3
-7	4	-1	-3
-7	-4	1	-3

Division by zero is considered a run-time error.

literal **LessEqual**

Less than or equal comparison binary operator (\leq).

literal **LessThan**

Less than comparison binary operator ($<$).

literal **Modulus**

Integer modulus binary operator (mod). Note that $a \text{ mod } b \equiv a - b \cdot (a \text{ div } b)$. The result has the same sign as the dividend (the first operand). For examples, see the *IntegerDivision* binary operator.

It is considered a run-time error if the second operand evaluates to zero.

literal **Multiplication**

Multiplication binary operator ($*$).

literal **Subset**

Subset binary operator (\subseteq).

literal **Subtraction**

Subtraction binary operator ($-$), as well as set difference (\setminus), and dictionary update.

literal **Unequal**

Inequality binary operator (\neq).

3.6.2 StdLibFunction (enumeration)

Standard library functions. Parameter types and return types are listed in the *StdLibFunction-Expression* (Section 3.6.36) class documentation.

literal **Abs**

Absolute value function: $abs(x) = |x|$.

literal **Acos**

Arc cosine function, with a result in the range $[0 .. \pi]$.

It is considered a run-time error if evaluation of the absolute value of the argument results in a number larger than one.

literal **Acosh**

Inverse hyperbolic cosine function.

literal **Asin**

Arc sine function, with a result in the range $[-\pi/2 .. \pi/2]$.

It is considered a run-time error if evaluation of the absolute value of the argument results in a number larger than one.

literal **Asinh**

Inverse hyperbolic sine function.

literal **Atan**

Arc tangent function, with a result in the range $[-\pi/2 .. \pi/2]$.

literal **Atanh**

Inverse hyperbolic tangent function.

literal **Bernoulli**

Bernoulli distribution function.

literal **Beta**

Beta distribution function.

literal **Binomial**

Binomial distribution function.

literal **Cbrt**

Cubic root function: $cbrt(x) = \sqrt[3]{x}$.

literal **Ceil**

Ceiling function. Rounds up (towards ∞). In other words, results in the smallest integer value that is not less than the argument.

literal **Constant**

Constant distribution function (useful for debugging/testing).

literal **Cos**

Cosine function, with angle given in radians.

literal **Cosh**

Hyperbolic cosine function: $\cosh(x) = (e^x + e^{-x})/2$.

literal **Delete**

Delete function, to remove an element from a list, by using a zero-based index. Negative indices count from the end of the list backwards.

It is considered a run-time error if an index is out of range for the list.

literal **Empty**

Empty function, to check whether containers are empty.

literal **Erlang**

Erlang distribution function.

literal **Exp**

Exponential function: $\exp(x) = e^x$.

literal **Exponential**

Exponential distribution function.

literal **Floor**

Floor function. Rounds down (towards $-\infty$). In other words, results in the largest integer value that is not exceed the argument.

literal **Format**

Formatting function. Applies a format pattern (first argument, string literal) to the remaining arguments.

literal **Gamma**

Gamma distribution function.

literal **Geometric**

Geometric distribution function.

literal **Ln**

Natural logarithmic function.

It is considered a run-time error if evaluation of the argument results in a non-positive number.

literal **Log**

Logarithmic (base 10) function.

It is considered a run-time error if evaluation of the argument results in a non-positive number.

literal **LogNormal**

Log-normal distribution function.

literal **Maximum**

Maximum value function: $\max(x, y) = \begin{cases} x & \text{if } x \geq y, \\ y & \text{if } x < y. \end{cases}$

literal **Minimum** (default)

Minimum value function: $\min(x, y) = \begin{cases} x & \text{if } x \leq y, \\ y & \text{if } x > y. \end{cases}$

literal **Normal**

Normal distribution function.

literal **Poisson**

Poisson distribution function.

literal **Pop**

Pops the first element from a list, and returns a tuple of the element and the list without that element.

It is considered a run-time error if the list is empty.

literal **Power**

Power (exponentiation) function: $\text{pow}(a, b) = a^b$. Note that $0^0 = 1$.

It is considered a run-time error if one of the following conditions holds during evaluation:

- the base is zero, and the exponent is negative,
- the base is negative, and the exponent is a non-integer number.

literal **Random**

Random (standard uniform) distribution function.

literal **Round**

Round function. Rounds to the nearest integer value. If the value is exactly between two integer values, it is rounded up (towards ∞). That is: $\text{round}(x) = \lfloor x + 0.5 \rfloor$.

literal **Scale**

Linear scaling with offset. A value v is interpreted in input interval $[inmin .. inmax]$, and transformed to a value in output interval $[outmin .. outmax]$:

$\text{scale}(v, inmin, inmax, outmin, outmax) = outmin + fraction * (outmax - outmin)$
with $fraction = (v - inmin) / (inmax - inmin)$

The intervals may be increasing or decreasing. The input value does not have to be included in the input interval. In that case it helps to think of the function as applying a linear transformation.

It is considered a run-time error if the input interval is empty ($inmin = inmax$), as that leads to division by zero.

literal **Sign**

Sign (or signum) function: $\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$

literal **Sin**

Sine function, with angle given in radians.

literal **Sinh**

Hyperbolic sine function: $\sinh(x) = (e^x - e^{-x})/2$.

literal **Size**

Function to get the size of a string, list, set, or dictionary.

literal **Sqrt**

Square root function: \sqrt{x} .

It is a run-time error if the argument evaluates to a negative number.

literal **Tan**

Tangent function, with angle given in radians.

literal **Tanh**

Hyperbolic tangent function: $\tanh(x) = \sinh(x)/\cosh(x)$.

literal **Triangle**

Triangle distribution function.

literal **Uniform**

Uniform distribution function.

literal **Weibull**

Weibull distribution function.

3.6.3 UnaryOperator (enumeration)

Unary operators for the *UnaryExpression* (Section 3.6.43). Type constraints are listed in the *UnaryExpression* (Section 3.6.43) class documentation.

literal **Inverse** (default)

Logical inverse unary operator (\neg).

literal **Negate**

Numerical negation (or additive inverse, or opposite) unary operator ($-$).

literal **Plus**

Numerical plus unary operator ($+$). Identity operator.

literal **Sample**

Sample unary operator. Draws a sample from a stochastic distribution, resulting in a tuple, with the sampled value, and the original distribution with a modified seed.

3.6.4 AlgVariableExpression (class)

An algebraic variable reference expression.

Constraints:

- **AlgVariableExpression.type** The type of this expression must match the type of the referenced algebraic variable. For ranged types, the range of the type of this expression must be equal to the range of the type of the algebraic variable, if it is specified.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *Expression* (Section 3.6.21)
 - ⊆ *AlgVariableExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **variable** [1] : *AlgVariable*
The referenced algebraic variable.

Constraints:

- **AlgVariableExpression.variableInScope** The algebraic variable reference must satisfy the scoping rules.

3.6.5 BaseFunctionExpression (abstract class)

Base class for function reference expressions.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *Expression* (Section 3.6.21)
 - ⊆ *BaseFunctionExpression*

Direct derived classes: *FunctionExpression* (Section 3.6.24), *StdLibFunctionExpression* (Section 3.6.36)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

3.6.6 BinaryExpression (class)

A binary expression.

Constraints:

- **BinaryExpression.type** The types of the left child expression, right child expression, and the type of the binary expression itself (the result type), depend on the *BinaryOperator* (Section 3.6.1). Table 3.1 lists the allowed combinations.

Note that for the *BinaryOperator.Equal* (Section 3.6.1) and *BinaryOperator.Unequal* (Section 3.6.1) binary operators, the types of the left and right child expressions must be equal. However, component types and component definition types, as well as function and distribution types, are not allowed, as those types don't support value equality. For ranged types, the ranges are ignored.

Note that for the *BinaryOperator.ElementOf* (Section 3.6.1) binary operator, the type of the left child expressions must not be a component type, a component definition type, a function type, or a distribution types, as those types don't support value equality. For ranged types, the ranges are ignored.

For the *BinaryOperator.Modulus* (Section 3.6.1) binary operator, the result range calculation is simplified, leading not only to simpler calculations, but also to more intuitive result ranges. It does however result in larger result ranges (over-approximations).

For integer types, if one of the operands has a rangeless integer type, the result type is also rangeless, if the result type is an integer type as well.

For list types, if one of the operands is an array, we try to keep the result an array as well. However, over approximations are sometimes used for the result types.

- **BinaryExpression.divideByZero** For the *BinaryOperator.IntegerDivision* (Section 3.6.1) and *BinaryOperator.Modulus* (Section 3.6.1) binary operators, if the right child expression has an integer type with range $[0 \dots 0]$, then the model is invalid, as this can only result in a division by zero run-time evaluation error.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
 - ⊢ *Expression* (Section 3.6.21)
 - ⊢ *BinaryExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

cont **left** [1] : *Expression*
The left child of the binary expression.

attr **operator** [1] : *BinaryOperator*
The binary operator of the binary expression.

cont **right** [1] : *Expression*
The right child of the binary expression.

Operator	Left type	Right type	Result type
Implication	bool	bool	bool
BiConditional	bool	bool	bool
Disjunction	bool	bool	bool
	set t	set t	set t
Conjunction	bool	bool	bool
	set t	set t	set t
LessThan	int / real	int / real	bool
LessEqual	int / real	int / real	bool
GreaterThan	int / real	int / real	bool
GreaterEqual	int / real	int / real	bool
Equal	t	t	bool
Unequal	t	t	bool
Addition	int $[l_1 .. u_1]$	int $[l_2 .. u_2]$	int $[l_1 + l_2 .. u_1 + u_2]$
	int	int	int
	int	real	real
	real	int	real
	real	real	real
	list t	list t	list t
	list $[l_1 .. u_1]$ t	list $[l_2 .. u_2]$ t	list $[l_1 + l_2 .. u_1 + u_2]$ t
	string	string	string
	dict(k:v)	dict(k:v)	dict(k:v)
Subtraction	int $[l_1 .. u_1]$	int $[l_2 .. u_2]$	int $[l_1 - u_2 .. u_1 - l_2]$
	int	int	int
	int	real	real
	real	int	real
	real	real	real
	set t	set t	set t
	dict(k:v)	dict(k:v)	dict(k:v)
	dict(k:v)	set k	dict(k:v)
	dict(k:v)	list k	dict(k:v)
	dict(k:v)	list $[l .. u]$ k	dict(k:v)
Multiplication	int $[l_1 .. u_1]$	int $[l_2 .. u_2]$	int $\left[\min \begin{pmatrix} l_1 \cdot l_2, \\ l_1 \cdot u_2, \\ u_1 \cdot l_2, \\ u_1 \cdot u_2 \end{pmatrix} .. \max \begin{pmatrix} l_1 \cdot l_2, \\ l_1 \cdot u_2, \\ u_1 \cdot l_2, \\ u_1 \cdot u_2 \end{pmatrix} \right]$
	int	int	int
	int	real	real
	real	int	real
	real	real	real
Division	int / real	int / real	real
IntegerDivision	int $[l_1 .. u_1]$	int $[l_2 .. u_2]$	int $\left[\min\{x \text{ div } y \mid x \in [l_1 .. u_1], y \in [l_2 .. u_2], y \neq 0\} .. \max\{x \text{ div } y \mid x \in [l_1 .. u_1], y \in [l_2 .. u_2], y \neq 0\} \right]$
	int	int	int
Modulus	int $[l_1 .. u_1]$	int $[l_2 .. u_2]$	int $\left[\begin{pmatrix} \min(0, -\max(l_2 , u_2) + 1) & \text{if } l_1 < 0 \\ 0 & \text{if } l_1 \geq 0 \end{pmatrix} .. \begin{pmatrix} \max(0, \max(l_2 , u_2) - 1) & \text{if } u_1 \geq 0 \\ 0 & \text{if } u_1 < 0 \end{pmatrix} \right]$
	int	int	int
ElementOf	t	list t	bool
	t	list $[l .. u]$ t	bool
	t	set t	bool
	k	dict(k:v)	bool
Subset	set t	set t	bool

Table 3.1: Binary expression type constraints.

3.6.7 BoolExpression (class)

A boolean value literal expression.

Constraints:

- **BoolExpression.type** The type of a boolean expression must be a boolean type.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *BoolExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)

The type of the expression.

attr **value** [1] : *EBoolean*

The boolean value.

3.6.8 CastExpression (class)

Cast expression.

It is considered a run-time error if casting from a string value, and the string value is not a valid textual representation of a value for the result type.

Constraints:

- **CastExpression.type** The types of the child expression and the cast expression itself must match. For t to t casts, the types must be exactly equal. That is, for ranged types the ranges must be equal as well. Table 3.2 lists the allowed combinations.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *CastExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

Child type	Cast/result type
int	real
int	string
real	string
bool	string
string	int (rangeless)
string	real
string	bool
automaton reference (including 'self')	string
t	t

Table 3.2: Cast expression type constraints.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

cont **child** [1] : *Expression*
The child of the cast expression.

3.6.9 CompInstWrapExpression (class)

Component instantiation reference wrapping expression. Allows keeping track of references via component instantiations. Similar to the *CompInstWrapType* (Section 3.5.3) class, but for expression references instead of type references. Similar to the *CompParamWrapExpression* (Section 3.6.11) class, but for references via component instantiations instead of via component parameters. See also Section 3.1.

Constraints:

- **CompInstWrapExpression.type** The type of the component instantiation reference wrapping expression must be equal to the type of its reference expression.

EObject

⊆ *PositionObject* (Section 3.11.2)
⊆ *Expression* (Section 3.6.21)
⊆ *CompInstWrapExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **instantiation** [1] : *ComponentInst*
The component instantiation via which the object is referenced.
Constraints:

- **CompInstWrapExpression.noCompDefBody** Components instantiations that are bodies of component definitions may not be referenced here, as the body *is* the component definition.
- **CompInstWrapExpression.instantiationInScope** The component instantiation reference must satisfy the scoping rules.

cont **reference** [1] : *Expression*

The object that is referenced via a component instantiation.

This feature contains *Expression* (Section 3.6.21) instances, to allow for wrapping expressions to be used.

Constraints:

- **CompInstWrapExpression.reference** The *reference* expression must be a reference.

3.6.10 CompParamExpression (class)

A component parameter reference expression.

Constraints:

- **CompParamExpression.type** The type of the component parameter reference expression must match the type of the referenced component parameter.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *CompParamExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **parameter** [1] : *ComponentParameter*
The referenced component parameter.

Constraints:

- **CompParamExpression.parameterInScope** The component parameter must satisfy the scoping rules.

3.6.11 CompParamWrapExpression (class)

Component parameter reference wrapping expression. Allows keeping track of references via component parameters. Similar to the *CompParamWrapType* (Section 3.5.4) class, but for expression references instead of type references. Similar to the *CompInstWrapExpression* (Section 3.6.9) class, but for references via component parameters instead of via component instantiations. See also Section 3.1.

Constraints:

- **CompParamWrapExpression.type** The type of the component parameter reference wrapping expression must be equal to the type of its reference expression.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
 - ⊆ *Expression* (Section 3.6.21)
 - ⊆ *CompParamWrapExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **parameter** [1] : *ComponentParameter*
The component parameter via which the object is referenced.
Constraints:

- **CompParamWrapExpression.parameterInScope** The component parameter reference must satisfy the scoping rules.

cont **reference** [1] : *Expression*
The object that is referenced via a component parameter.

This feature contains *Expression* (Section 3.6.21) instances, to allow for wrapping expressions to be used.

Constraints:

- **CompParamWrapExpression.reference** The *reference* expression must be a reference.

3.6.12 ComponentExpression (class)

A component reference expression.

Constraints:

- **ComponentExpression.type** The type of the component reference expression must be a component type that refers to the same component as this expression does.

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *Expression* (Section 3.6.21)
 ⊆ *ComponentExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
 The type of the expression.

ref **component** [1] : *Component*
 The referenced component.

Constraints:

- **ComponentExpression.noCompDefBody** Components that are bodies of component definitions may not be referenced here, as the body *is* the component definition.
- **ComponentExpression.noSpec** Specifications, which are technically components, may not be referenced here, as they serve as outer grouping only. Note that in the CIF textual syntax, they can't be referenced either, as they don't have a name.
- **ComponentExpression.componentInScope** The component reference must satisfy the scoping rules.

3.6.13 ConstantExpression (class)

A constant reference expression.

Constraints:

- **ConstantExpression.type** The type of this expression must match the type of the referenced constant. For ranged types, the range of the type of this expression must be equal to the range of the type of the constant, if specified.

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *Expression* (Section 3.6.21)
 ⊆ *ConstantExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **constant** [1] : *Constant*
The referenced constant.

Constraints:

- **ConstantExpression.constantInScope** The constant reference must satisfy the scoping rules.

3.6.14 ContVariableExpression (class)

A continuous variable reference expression.

Constraints:

- **ContVariableExpression.type** The type of the variable reference expression must be a real type.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *Expression* (Section 3.6.21)

⊢ *ContVariableExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

attr **derivative** [1] : *EBoolean*
Indicates whether this reference is a reference to the derivative of the variable (*true*) or to the variable itself (*false*).

ref **variable** [1] : *ContVariable*
The referenced continuous variable.

Constraints:

- **ContVariableExpression.variableInScope** The continuous variable reference must satisfy the scoping rules.

3.6.15 DictExpression (class)

A dictionary literal expression.

It is considered a run-time error if a dictionary literal has duplicate keys.

Constraints:

- **DictExpression.type** The type of the dictionary expression must be a dictionary type. Each of the keys of the pairs must match the key type of the dictionary type. Each of the values of the pairs must match the value type of the dictionary type. For ranged types, the ranges (if specified) of the keys and values must be contained in the range of the key type and value type respectively.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *DictExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

cont **pairs** [0..*] : *DictPair*
The key/value pairs of the dictionary.

3.6.16 DictPair (class)

A single key/value pair for a *DictExpression* (Section 3.6.15).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *DictPair*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **key** [1] : *Expression*
The key of the key/value pair.

cont **value** [1] : *Expression*
The value of the key/value pair.

3.6.17 DiscVariableExpression (class)

A discrete variable reference expression.

Constraints:

- **DiscVariableExpression.type** The type of the discrete variable reference expression must match the type of the referenced discrete variable. For ranged types, the ranges (if specified) must be equal.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *Expression* (Section 3.6.21)
 - ⊢ *DiscVariableExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **variable** [1] : *DiscVariable*
The referenced discrete variable.

Constraints:

- **DiscVariableExpression.variableInScope** The discrete variable reference must satisfy the scoping rules.

3.6.18 ElifExpression (class)

An ‘elif’ (‘else-if’) alternative of an *IfExpression* (Section 3.6.25).

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *ElifExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **guards** [1..*] : *Expression*
The guard predicates for this alternative.

If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between the guards.

Constraints:

- **ElifExpression.guardTypes** The guard predicates must have boolean types.

cont **then** [1] : *Expression*

The value that the *IfExpression* (Section 3.6.25) evaluates to if the ‘guard’ evaluates to *true*.

3.6.19 EnumLiteralExpression (class)

An enumeration literal reference expression.

Constraints:

- **EnumLiteralExpression.type** The type of an enumeration literal expression must be equal to the enumeration that the literal is a part of.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *EnumLiteralExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **literal** [1] : *EnumLiteral*
The referenced enumeration literal.

Constraints:

- **EnumLiteralExpression.literalInScope** The enumeration literal reference must satisfy the scoping rules.

3.6.20 EventExpression (class)

An event reference expression.

Only used for event references, as events can’t be used in a value context. It is part of the expression tree, to allow wrapping expressions to be used. See also Section 3.1.

Can not be used to refer to the ‘tau’ event.

Constraints:

- **EventExpression.type** The type of an event expression must be a boolean type.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *Expression* (Section 3.6.21)
- ⊢ *EventExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **event** [1] : *Event*
The referenced event.

Constraints:

- **EventExpression.eventInScope** The event reference must satisfy the scoping rules.
- **EventExpression.occurrence** Events may only be used in places where they are directly used as event reference (such as events on edges, in alphabets, in monitor sets, in component instantiation arguments for event parameters, etc). They may thus explicitly not be used in functions, guards, initial values of variables, etc.

3.6.21 Expression (abstract class)

Base class for all CIF expressions.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *Expression*

Direct derived classes: *AlgVariableExpression* (Section 3.6.4), *BaseFunctionExpression* (Section 3.6.5), *BinaryExpression* (Section 3.6.6), *BoolExpression* (Section 3.6.7), *CastExpression* (Section 3.6.8), *CompInstWrapExpression* (Section 3.6.9), *CompParamExpression* (Section 3.6.10), *CompParamWrapExpression* (Section 3.6.11), *ComponentExpression* (Section 3.6.12), *ConstantExpression* (Section 3.6.13), *ContVariableExpression* (Section 3.6.14), *DictExpression* (Section 3.6.15), *DiscVariableExpression* (Section 3.6.17), *EnumLiteralExpression* (Section 3.6.19), *EventExpression* (Section 3.6.20), *FieldExpression* (Section 3.6.22), *FunctionCallExpression* (Section 3.6.23), *IfExpression* (Section 3.6.25), *InputVariableExpression* (Section 3.6.26), *IntExpression* (Section 3.6.27), *ListExpression* (Section 3.6.28), *LocationExpression* (Section 3.6.29), *ProjectionExpression* (Section 3.6.30), *RealExpression* (Section 3.6.31), *ReceivedExpression* (Section 3.6.32), *SelfExpression*

(Section 3.6.33), *SetExpression* (Section 3.6.34), *SliceExpression* (Section 3.6.35), *StringExpression* (Section 3.6.37), *SwitchExpression* (Section 3.6.39), *TauExpression* (Section 3.6.40), *TimeExpression* (Section 3.6.41), *TupleExpression* (Section 3.6.42), *UnaryExpression* (Section 3.6.43)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType*
The type of the expression.

3.6.22 FieldExpression (class)

A tuple field reference expression.

Constraints:

- **FieldExpression.occurrence** Instances of *FieldExpression* may only occur in the *index* feature of the *ProjectionExpression* (Section 3.6.30) class. They must not be wrapped in other expressions.
- **FieldExpression.type** The type of a field expression must be an integer type, with as range the single value that corresponds to the 0-based index of the field in the tuple type of which it is a part.

EObject

⊆ *PositionObject* (Section 3.11.2)
⊆ *Expression* (Section 3.6.21)
⊆ *FieldExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **field** [1] : *Field*
The referenced tuple field.

Constraints:

- **FieldExpression.fieldInScope** The field reference must satisfy the scoping rules. For the details, see the *ProjectionExpression* (Section 3.6.30).

3.6.23 FunctionCallExpression (class)

A function call expression.

Constraints:

- **FunctionCallExpression.type** The type of the function call expression must match the types of the function and its parameters. That is, the function must have a function type. The type of the functional call expression must match the result type of the function type. For ranged types, the ranges (if specified) must be equal. The count and types of the arguments must match the parameter types of the function type. For ranged types, the ranges (if specified) of the arguments must be contained in the ranges of the parameter types of the function type.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *FunctionCallExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

cont **arguments** [0..*] : *Expression*
The function arguments for the function call.

cont **function** [1] : *Expression*
The function to call.

3.6.24 FunctionExpression (class)

A user-defined function reference expression.

Constraints:

- **FunctionExpression.type** The type of a function reference expression must be a function type, with a return type matching the return type of the referenced function, and parameter types matching the types of the parameters of the referenced function. For ranged types, the ranges must match exactly.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *BaseFunctionExpression* (Section 3.6.5)

⊆ *FunctionExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

ref **function** [1] : *Function*
The referenced user-defined function.
Constraints:

- **FunctionExpression.functionInScope** The function reference must satisfy the scoping rules.

3.6.25 IfExpression (class)

A conditional expression.

Constraints:

- **IfExpression.type** The type of the ‘then’ expression, the type of the ‘else’ expression, the types of the ‘thens’ of the ‘elif’ expressions (if any), and the type of this expression, must all match.

Component and component definition types are not allowed.

For integer types, for ‘then’ range $[lt .. ut]$, for ‘else’ range $[le .. ue]$, for ‘elif’ ranges $[lf_n .. uf_n]$, in case of n ‘elif’ alternatives, $0 < n$, the range of this expression must be equal to $[lt \min le \min \left(\min_{0 \leq i < n} (lf_i) \right) .. ut \max ue \max \left(\max_{0 \leq i < n} (uf_i) \right)]$. That is, the ranges are merged. Similarly, ranges for other ranged types are merged.

Note that the guards of the alternatives are not taken into account. Also, if one of the alternatives has a non-ranged integer type, the result has a non-ranged integer type as well. Similar notes apply to other ranged types.

EObject

⊢ *PositionObject* (Section 3.11.2)
⊢ *Expression* (Section 3.6.21)
⊢ *IfExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

cont **elifs** [0..*] : *ElifExpression*
The ‘elif’ (‘else-if’) alternatives. Processed in order, if the ‘guard’ evaluates to *false*.

cont **else** [1] : *Expression*

The ‘else’ value. Returned if no other alternative has a *true* guard.

cont **guards** [1..*] : *Expression*

The guard predicates for the ‘then’ value.

If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between the guards.

Constraints:

- **IfExpression.guardTypes** The guard predicates must have boolean types.

cont **then** [1] : *Expression*

The ‘then’ value. Returned if the ‘guard’ evaluates to *true*.

3.6.26 InputVariableExpression (class)

An input variable reference expression.

Constraints:

- **InputVariableExpression.type** The type of this expression must match the type of the referenced input variable. For ranged types, the ranges (if specified) must be equal.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *InputVariableExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)

The type of the expression.

ref **variable** [1] : *InputVariable*

The referenced input variable.

Constraints:

- **InputVariableExpression.variableInScope** The input variable reference must satisfy the scoping rules.

3.6.27 IntExpression (class)

An integer value literal expression.

The value of integer value literal expressions is encoded using the *EInt* datatype, which matches 32-bit integers in the range $[-2^{31} .. 2^{31} - 1]$. Note that integer values are also used for the bounds of ranges (see *IntType* (Section 3.5.12)), which means that both values and integer type ranges are limited to that given range of integers.

Operations that result in overflow or underflow should be considered errors, and under no circumstance should wrapping or saturation take place.

If operations depend only on integer types with ranges, we enforce that no overflow can ever happen, by using static semantic constraints, as to avoid run-time overflow detection. If one of the operands involves a rangeless integer type, range checking is disabled for that operation, deferring the responsibility to run-time, and thus the implementation.

Constraints:

- **IntExpression.type** The type of an integer expression with value v must be an integer type with range $[v .. v]$.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *Expression* (Section 3.6.21)
- ⊆ *IntExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

attr **value** [1] : *EInt*
The integer value. May be negative, zero, or positive.

3.6.28 ListExpression (class)

A list literal expression.

Operations that result in overflow on the sizes of lists should be considered errors.

If operations depend only on list types with ranges, we enforce that no out-of-range errors can ever happen, by using static semantic constraints, as to avoid run-time out-of-range detection. If one of the operands involves a rangeless list type, range checking is disabled for that operation, deferring the responsibility to run-time, and thus the implementation.

Constraints:

- **ListExpression.type** The type of the list expression must be a list type, with range $[n .. n]$ for a list of n elements. Each of the elements must match the element type of the list type. For ranged element types, the ranges (if specified) of the types of the elements must be contained in the range of the element type of the list type.

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *Expression* (Section 3.6.21)
 ⊆ *ListExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
 The type of the expression.

cont **elements** [0..*] : *Expression*
 The elements of the list.

3.6.29 LocationExpression (class)

A location reference expression. Represents an ‘is the location active’ boolean expression.

Constraints:

- **LocationExpression.type** The type of a location expression must be a boolean type.

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *Expression* (Section 3.6.21)
 ⊆ *LocationExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
 The type of the expression.

ref **location** [1] : *Location*
 The referenced location.

Constraints:

- **LocationExpression.locationInScope** The location reference must satisfy the scoping rules.

Child type	Index (type)	Projection/result type
list t	int	t
list t	int[a..b]	t
list [l .. u] t	int	t
list [l .. u] t	int[a..b]	t
dict(k:v)	k	v
string	int	string
tuple(t1 f1, t2 f2, ..., tn fn)	int i	ti
tuple(t1 f1, t2 f2, ..., tn fn)	field fi	ti

Table 3.3: Projection expression type constraints.

3.6.30 ProjectionExpression (class)

Projection expression.

Lists can be indexed using a zero-based index. Negative indices count from the end of the list backwards. It is considered a run-time error if an index is out of range for the list.

Dictionaries can be indexed using keys. It is considered a run-time error if a key is not in the dictionary. An exception is the last projection of an addressable, as a new key/value pair is created if the key does not exist for such a projection.

Strings can be indexed similar to lists.

Tuples can be indexed using a zero-based index. It is considered a run-time error if an index is out of range (negative or greater than or equal to the size of the tuple).

Tuples can also be indexed using field names. In such a case, the *index* expression must be a *FieldExpression* (Section 3.6.22), which must not be wrapped by any other expressions.

Constraints:

- **ProjectionExpression.type** The types of the child expression and the projection expression itself must match. Table 3.3 lists the allowed combinations.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *ProjectionExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

cont **child** [1] : *Expression*
The child expression of the projection expression. This is the value that is being projected.

cont **index** [1] : *Expression*

The projection index of the projection expression. This indicates what to project from the child expression.

Constraints:

- **ProjectionExpression.indexInScope** For projection on tuples, the fields of the tuple type of the child expression are in scope, and the fields take precedence over any other objects with those names, in the same scope.

3.6.31 RealExpression (class)

A real number value literal expression.

Constraints:

- **RealExpression.type** The type of a real expression must be a real type.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *RealExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)

The type of the expression.

attr **value** [1] : *EString*

The real number value. Note that real numbers are represented in the metamodel as strings, to keep the original formatting, and to keep the precise value. For details on the representation of real numbers in the implementation, see the *RealType* (Section 3.5.14) class.

The syntax of the strings is CIF textual syntax for real numbers. Note that this explicitly excludes negative real numbers, which are real numbers with a unary negation operator.

3.6.32 ReceivedExpression (class)

A reference to the value received as part of a channel communication.

Constraints:

- **ReceivedExpression.scope** The received value only exists in the updates of edges, for edges where a value is received. Received values may not be assigned (are read-only). They may be used on the right hand side of assignments, as projection indices in the addressables, and in guards for ‘if’ updates.

- **ReceivedExpression.type** The type of a received value reference expression must have the same type as the type of the event or events over which data is communicated. If multiple events are present on the edge, they must have equal types. If communication over a *VoidType* (Section 3.5.19) channel is performed, the received value does not exist, as there are no values for ‘void’ types.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *Expression* (Section 3.6.21)
- ⊆ *ReceivedExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

3.6.33 SelfExpression (class)

A self reference to an automaton or automaton definition.

Constraints:

- **SelfExpression.scope** The self expression may only be used within automata and automaton definitions.
- **SelfExpression.type** The type of a self expression must be an automaton type or automaton definition type, referring to the automaton or automaton definition in which the self expression is used.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *Expression* (Section 3.6.21)
- ⊆ *SelfExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

3.6.34 SetExpression (class)

A set literal expression.

Constraints:

- **SetExpression.type** The type of the set expression must be a set type. Each of the elements must match the element type of the set type. For ranged element types, the ranges (if specified) of the types of the elements must be contained in the range of the element type of the set type.

EObject

⊢ *PositionObject* (Section 3.11.2)
⊢ *Expression* (Section 3.6.21)
⊢ *SetExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

cont **elements** [0..*] : *Expression*
The elements of the set.

3.6.35 SliceExpression (class)

Slice expression.

Lists can also be sliced, using a *begin* and *end* index. This results in a part of the original list, for the sub-range [*begin* .. *end* − 1]. Both indices are optional. An omitted *begin* index equals 0, and an omitted *end* index equals the length of the list. Indices may be negative to count from the end of the list backwards. Out of range slice indices are handled gracefully. An index that is too large is replaced by the list size. A lower bound larger than the upper bound results in an empty list. This applies to negative indices as well.

Strings can be sliced similar to lists.

Constraints:

- **SliceExpression.type** The types of the child expression and the slice expression itself must match. Table 3.4 lists the allowed combinations.

EObject

⊢ *PositionObject* (Section 3.11.2)

Child type	begin	end	Slice/result type
list t	int/int [a .. b]/omit	int/int [a .. b]/omit	list t
list [l .. l] t	int/int [y .. y]/omit	int/int [z .. z]/omit	list [w .. w], with w only possible result size
list [l .. l] t	int (other)	int (other)	list [0 .. l]
list [l .. u] t	int/int [y .. y]/omit, with 0 <= y <= l	int/int [z .. z]/omit, with 0 <= z <= l	list [v .. w], with v and w as narrow as possible
list [l .. u] t	int (other)	int (other)	list [0 .. u]
string	int	int	string

Table 3.4: Slice expression type constraints.

⊢ *Expression* (Section 3.6.21)
 ⊢ *SliceExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
 The type of the expression.

cont **begin** [0..1] : *Expression*
 The begin index. May be omitted to default to 0.

cont **child** [1] : *Expression*
 The child expression. This is the value that is being sliced.

cont **end** [0..1] : *Expression*
 The end index. May be omitted to default to the length of the list.

3.6.36 StdLibFunctionExpression (class)

Standard library function reference expression.

Constraints:

- **StdLibFunctionExpression.occurrence** Standard library function references may only occur as the ‘function’ of a *FunctionCallExpression* (Section 3.6.23). They must occur there directly, as the value of the ‘function’ feature, without any other expressions around them. This is mainly to ensure that distribution standard library functions can not be passed around as values.
- **StdLibFunctionExpression.occurrenceDist** Standard library function references for distribution functions, may only in the ‘value’ of the *DiscVariable* (Section 3.3.5) class. That is, distributions may only be created in the initial values of discrete variables, declared in automata.

- **StdLibFunctionExpression.type** The type of the standard library expression depends on the *StdLibFunction* (Section 3.6.2), and for some functions, also on the arguments. Tables 3.5, 3.6, and 3.7 list the allowed combinations.

For integer types, if one of the arguments has a rangeless integer type, the result type is also rangeless, if the result type is an integer type as well. Similar relations hold for other ranged types.

The *StdLibFunction.Format* (Section 3.6.2) function has as first argument a string typed value, and may additionally have more arguments, of any type.

- **StdLibFunctionExpression.formatPatternLiteral** The first argument to the *StdLibFunction.Format* (Section 3.6.2) function must be a string literal expression, to use as format pattern.
- **StdLibFunctionExpression.formatPattern** The format pattern of the *StdLibFunction.Format* (Section 3.6.2) function must not have any decoding errors, explicit indices must not result in integer overflow, used indices (1-based, either implicit or explicit) may not be out of range, and the values corresponding to the specifiers must have appropriate types.
- **StdLibFunctionExpression.formatUsed (non-fatal)** The format pattern (first argument) of the *StdLibFunction.Format* (Section 3.6.2) function should contain specifiers with indices (1-based, either implicit or explicit) that refer to each of the other arguments of the function call.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *Expression* (Section 3.6.21)
 - ⊆ *BaseFunctionExpression* (Section 3.6.5)
 - ⊆ *StdLibFunctionExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

attr **function** [1] : *StdLibFunction*
The referenced standard library function.

3.6.37 StringExpression (class)

A string value literal expression.

Constraints:

- **StringExpression.type** The type of a string expression must be a string type.

Function	Argument types	Result type
Abs	int $[l .. u]$	int $\left[\begin{array}{l} \min\{ x \mid x \in [l .. u]\} .. \\ \max\{ x \mid x \in [l .. u]\} \end{array} \right]$
	int	int
	real	real
Cbrt	real	real
Ceil	real	int
Delete	list t, int	list t
	list t, int[a..b]	list t
	list $[l .. u]$ t, int	list $[0 \max l - 1 .. 0 \max u - 1]$ t
	list $[l .. u]$ t, int[a..b]	list $[0 \max l - 1 .. 0 \max u - 1]$ t
Empty	list t	bool
	list $[l .. u]$ t	bool
	set t	bool
	dict(k:v)	bool
Exp	real	real
Floor	real	int
Format	string, ...	string
Ln	real	real
Log	real	real
Minimum	int $[l_1 .. u_1]$, int $[l_2 .. u_2]$	int $[l_1 \min l_2 .. u_1 \min u_2]$
	int, int	int
	real, int	real
	int, real	real
	real, real	real
Maximum	int $[l_1 .. u_1]$, int $[l_2 .. u_2]$	int $[l_1 \max l_2 .. u_1 \max u_2]$
	int, int	int
	real, int	real
	int, real	real
	real, real	real
Pop	list t	tuple(t, list t)
	list $[l .. u]$ t	tuple(t, list $[0 \max l - 1 .. 0 \max u - 1]$ t)
Power	int $[l_1 .. u_1]$, int $[l_2 .. u_2]$, $l_2 \geq 0$	int $\left[\begin{array}{l} \min\{x^y \mid x \in [l_1 .. u_1], y \in [l_2 .. u_2]\} .. \\ \max\{x^y \mid x \in [l_1 .. u_1], y \in [l_2 .. u_2]\} \end{array} \right]$
	int / real, int / real	real
Round	real	int
Scale	int / real, int / real, ... (5 arguments)	real
Sign	int $[l .. u]$	int $\left[\begin{array}{l} \left(\begin{array}{ll} -1 & \text{if } l < 0 \\ 0 & \text{if } l = 0 \\ 1 & \text{if } l > 0 \end{array} \right) .. \left(\begin{array}{ll} -1 & \text{if } u < 0 \\ 0 & \text{if } u = 0 \\ 1 & \text{if } u > 0 \end{array} \right) \end{array} \right]$
	int	int $[-1 .. 1]$
	real	int $[-1 .. 1]$
Size	string	int
	list t	int
	list $[l .. u]$ t	int $[l .. u]$
	set t	int
	dict(k:v)	int
Sqrt	real	real

Table 3.5: General standard library function type constraints.

Function	Argument types	Result type
Acosh	real	real
Acos	real	real
Asinh	real	real
Asin	real	real
Atanh	real	real
Atan	real	real
Cosh	real	real
Cos	real	real
Sinh	real	real
Sin	real	real
Tanh	real	real
Tan	real	real

Table 3.6: Trigonometric standard library function type constraints.

Function	Argument types	Result type
Bernoulli	real	dist bool
Beta	real, real	dist real
Binomial	real, int	dist int
Constant	bool	dist bool
	int	dist int
	real	dist real
Erlang	int, real	dist real
Exponential	real	dist real
Gamma	real, real	dist real
Geometric	real	dist int
LogNormal	real, real	dist real
Normal	real, real	dist real
Poisson	real	dist int
Random	-	dist real
Triangle	real, real, real	dist real
Uniform	int, int	dist int
	real, real	dist real
Weibull	real, real	dist real

Table 3.7: Distribution standard library function type constraints.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *Expression* (Section 3.6.21)

⊢ *StringExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)

The type of the expression.

attr **value** [1] : *EString*

The string value.

3.6.38 SwitchCase (class)

A single case of a *SwitchExpression* (Section 3.6.39).

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *SwitchCase*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **key** [0..1] : *Expression*

The key of the switch case. If specified, the case can only be selected if the ‘value’ of the *SwitchExpression* (Section 3.6.39) is equal to this key. If not specified, the case is an ‘else’ case and it can always be selected.

This feature contains *Expression* (Section 3.6.21) instances. If the ‘value’ of the switch expression refers to an automaton, the key is a location reference. The expression is stored in such a way that it refers to the location from the scope of this expression. Wrapping expressions may thus be used. See also Section 3.1. In the textual syntax, only an identifier (possibly escaped) may be used in such cases.

Constraints:

- **SwitchCase.keyType** If the ‘key’ is specified, and if the ‘value’ of the switch expression does not refer to an automaton, the type of the ‘key’ expression and the type of that ‘value’ must be type compatible (ignoring ranges).
- **SwitchCase.keyLocRef** If the ‘key’ is specified, and if the ‘value’ of the switch expression refers to an automaton, the ‘key’ must be a reference to a location of that automaton. The reference must be valid from the scope of this expression. In the textual syntax, only an identifier (possibly escaped) may be used.

cont **value** [1] : *Expression*

The value of the switch case. The value is used as result of the switch expression if the case is selected.

Constraints:

- **SwitchCase.valueType** Component types and component definition types are not allowed for values of cases, as components and component definitions are not meant to be used as values.

3.6.39 SwitchExpression (class)

A switch expression.

Constraints:

- **SwitchExpression.type** The type of the switch expression must be the union of the types of the ‘value’ expressions of the ‘cases’. This implies that those expressions must be type compatible (ignoring ranges).
- **SwitchExpression.complete** The ‘cases’, possibly including an ‘else’, must be complete for all locations of the automaton, or all possible values of the type of the switch value otherwise.
- **SwitchExpression.overspecified** The ‘cases’ must not be overspecified (same location or value multiple times as *key* of one of the cases).
- **SwitchExpression.superfluousElse (non-fatal)** The ‘cases’ must not be overspecified (an ‘else’ is present while all locations or values are already specified as ‘key’ of a case).
- **SwitchExpression.singleCase (non-fatal)** There must be at least two cases (including any ‘else’ case, if present).

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *Expression* (Section 3.6.21)

⊢ *SwitchExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)

The type of the expression.

cont **cases** [1..*] : *SwitchCase*

The cases of the switch. They are processed in order.

Constraints:

- **SwitchExpression.elseOccurrence** At most one of the ‘cases’ may be an ‘else’. If it is present, it must be the last of the ‘cases’.
- **SwitchExpression.elseMandatory** If the ‘value’ does not refer to an automaton, and it can not be statically determined whether the switch expression is complete without an ‘else’ case, an ‘else’ case is mandatory.

cont **value** [1] : *Expression*

The control value of the switch expression.

Constraints:

- **SwitchExpression.valueEquality** If the ‘value’ does not refer to an automaton, it must have a type that supports value equality.
- **SwitchExpression.valueType** Component types and component definition types are only allowed if the ‘value’ refers to an automaton.

3.6.40 TauExpression (class)

A ‘tau’ event reference expression. Refers to the implicitly always present non-synchronizing ‘tau’ event. This event is neither controllable nor uncontrollable.

Constraints:

- **TauExpression.occurrence** Instances of *TauExpression* (Section 3.6.40) are only allowed in the ‘events’ feature of the *EdgeEvent* (Section 3.4.5) class.
- **TauExpression.type** The type of a ‘tau’ expression must be a boolean type, similar to event expressions.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Expression* (Section 3.6.21)

⊆ *TauExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)

The type of the expression.

3.6.41 TimeExpression (class)

A reference to the implicitly always available, global variable ‘time’.

Constraints:

- **TimeExpression.type** The type of a time reference expression must be a real type.

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *Expression* (Section 3.6.21)
 ⊆ *TimeExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.
 cont **type** [1] : *CifType* (inherited from *Expression*)
 The type of the expression.

3.6.42 TupleExpression (class)

A tuple literal expression.

Constraints:

- **TupleExpression.type** The type of the tuple expression must be a tuple type. Each of the fields must match the type of the corresponding field of the tuple type. For ranged types, the ranges (if specified) of the types of the fields (elements) must be equal to the range of the type of the corresponding field of the tuple type.

EObject

⊆ *PositionObject* (Section 3.11.2)
 ⊆ *Expression* (Section 3.6.21)
 ⊆ *TupleExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
 Optional position information.
 cont **type** [1] : *CifType* (inherited from *Expression*)
 The type of the expression.
 cont **fields** [2..*] : *Expression*
 The fields (elements) of the tuple.

3.6.43 UnaryExpression (class)

A unary expression.

Constraints:

Operator	Child type	Result type
Inverse	bool	bool
Negate	int [<i>l</i> .. <i>u</i>]	int [$-u$.. $-l$]
	int	int
	real	real
Plus	int [<i>l</i> .. <i>u</i>]	int [<i>l</i> .. <i>u</i>]
	int	int
	real	real
Sample	dist <i>t</i>	tuple(<i>t</i> , dist <i>t</i>)

Table 3.8: Unary expression type constraints.

- **UnaryExpression.type** The types of the child expression and the unary expression itself (the result type), depend on the *UnaryOperator* (Section 3.6.3). Table 3.8 lists the allowed combinations.

For integer types, if the child has a rangeless integer type, the result type is also rangeless, if the result type is an integer type as well. Similar relations hold for other ranged types.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *Expression* (Section 3.6.21)

⊢ *UnaryExpression*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **type** [1] : *CifType* (inherited from *Expression*)
The type of the expression.

cont **child** [1] : *Expression*
The child of the unary expression.

attr **operator** [1] : *UnaryOperator*
The unary operator of the unary expression.

3.7 Package functions

Figure 3.9 shows the *functions* package. The *Function* (Section 3.7.6) class acts as a base class for all user-defined functions. We have two concrete variants of user-defined functions: internal user-defined functions, which are fully defined within the CIF specification, and external user-defined function, for which the header is defined in the CIF specification, and the actual implementation resides outside of the CIF specification. The upper part of the diagram is mostly concerned with the features common to all user-defined functions. The lower part contains the statements that can be used in the internal user-defined functions.

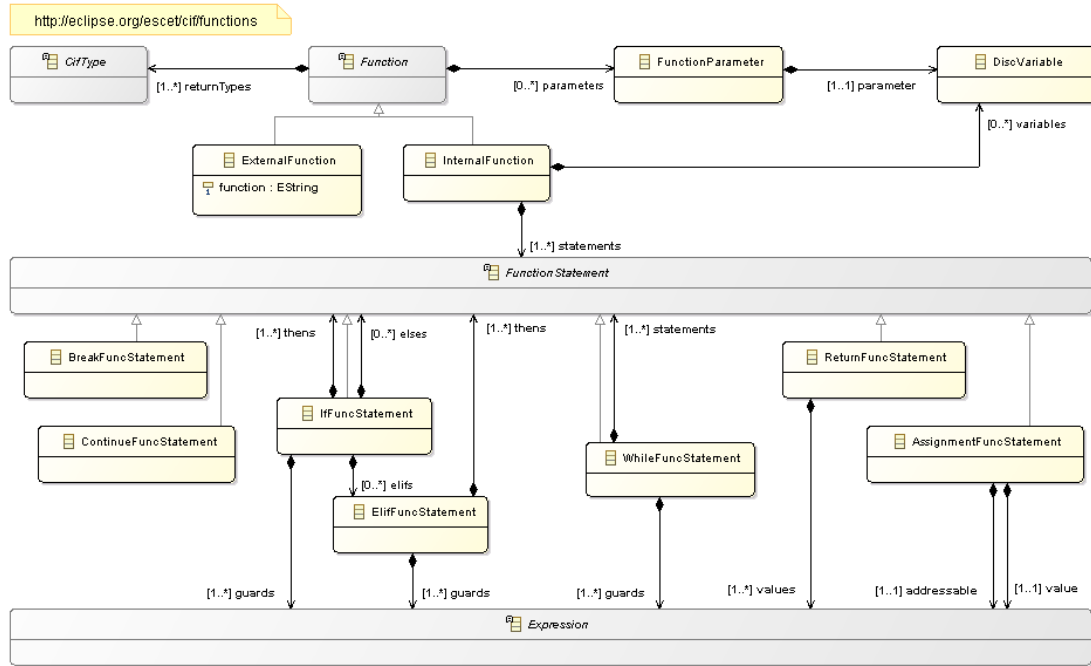


Figure 3.9: *functions* package

Package URI <http://eclipse.org/escet/cif/functions>

Namespace prefix functions

Sub-packages none

3.7.1 AssignmentFuncStatement (class)

An assignment internal function statement.

Constraints:

- **AssignmentFuncStatement.types** The type of the assigned value must match the type of the addressable that is assigned. For ranged types, if the variable being assigned has a ranged type with a range, then the range of the type of the value must have overlap with the range of the type of the variable. It is considered a run-time error if the evaluated value expression results in a value that is outside of the range of the assigned variable.

EObject

↳ *PositionObject* (Section 3.11.2)

↳ *FunctionStatement* (Section 3.7.8)

↳ *AssignmentFuncStatement*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **addressable** [1] : *Expression*

The addressable (a variable, a part of a variable, multiple variables, parts of multiple variables, etc) to which to assign a value.

It is allowed to create a new key/value pair in a dictionary, if the key of the last projection does not exist. It is not allowed to create a new key/value pair, if the key of one of the other projections does not exist.

Constraints:

- **AssignmentFuncStatement.addressableSyntax** Adressables may be discrete variable references (non-wrapped), with projections, and may optionally be wrapped in tuples (possibly multiple times). Projected string typed variables are not allowed.
- **AssignmentFuncStatement.variablesInScope** The variables that are assigned must be local variables declared in the same function as the assignment statement, or they must be parameters of that same function. Local variables and parameters may be mixed in a single assignment.
- **AssignmentFuncStatement.uniqueVariables** The parts of variables that are assigned must be unique. That is, it must never be possible to assign the same part of the same variable twice. We do include the projections in the analysis, but only as far as we can statically evaluate and normalize the indices. See the ‘Edge.uniqueVariables’ constraint for the complete details.

cont **value** [1] : *Expression*

The value to assign to the variables.

3.7.2 BreakFuncStatement (class)

A break internal function statement. Breaks out of the closest enclosing while statement.

Constraints:

- **BreakFuncStatement.occurrence** Break statements may only occur in the body of while statements.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *FunctionStatement* (Section 3.7.8)

⊆ *BreakFuncStatement*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

3.7.3 ContinueFuncStatement (class)

A continue internal function statement. Continues with the next iteration (if any) of the closest enclosing while statement, or breaks out of it (if the guards no longer hold).

Constraints:

- **ContinueFuncStatement.occurrence** Continue statements may only occur in the body of while statements.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *FunctionStatement* (Section 3.7.8)
 - ⊢ *ContinueFuncStatement*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

3.7.4 ElifFuncStatement (class)

An ‘elif’ (‘else-if’) alternative of an *IfFuncStatement* (Section 3.7.9).

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *ElifFuncStatement*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **guards** [1..*] : *Expression*
The guard predicates for this alternative.

If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between the guards.

Constraints:

- **ElifFuncStatement.guardTypes** The guard predicates must have boolean types.

cont **thens** [1..*] : *FunctionStatement*
The statements to execute for the *ElifFuncStatement* (Section 3.7.4) if the ‘guards’ evaluate to *true*.

3.7.5 ExternalFunction (class)

An external user-defined function. The CIF specification only defines the header of the function, and the actual implementation resides outside of the CIF specification.

External user-defined functions are generally used to allow access from within CIF specifications to algorithms and functions defined in other languages. Functions are used to perform calculations, rather than for parts of the system that have run-time behavior, such as discrete event control algorithms, or hybrid dynamics.

Constraints:

- **ExternalFunction.uniqueParams** The names of all parameters of the function, must be unique within that function.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
 - ⊢ *AnnotatedObject* (Section 3.10.1)
 - ⊢ *Declaration* (Section 3.3.4)
 - ⊢ *Function* (Section 3.7.6)
 - ⊢ *ExternalFunction*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)
The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**_, **c**_, or **u**_, as those names are reserved for events.

cont **parameters** [0..*] : *FunctionParameter* (inherited from *Function*)

The parameters of the user-defined function.

cont **returnTypes** [1..*] : *CifType* (inherited from *Function*)

The return types of the user-defined function. If multiple types are defined, then the return type of the function is a tuple with nameless fields of those types.

Constraints:

- **Function.allowedReturnTypes** Component types and component definition types are not allowed in return types of functions, as components and component definitions are not meant to be used as values.

attr **function** [1] : *EString*

A textual reference to the external implementation of the function. The contents is interpreted by the tooling, such as for instance simulators.

3.7.6 Function (abstract class)

Base class for all external user-defined functions.

We use value semantics for the parameters of functions.

Constraints:

- **Function.sideEffectFree** All user-defined functions in CIF are pure mathematical functions and therefore, must be deterministic, and may not have side effects.

This is enforced for internal user-defined functions by not allowing the use of variable ‘time’, the scoping rules regarding references to objects outside of the function, and the restricted use of distribution standard library functions.

For external user-defined functions, it is (often) impossible to check this constraint in an implementation, and the responsibility for checking this is therefore delegated to the end user. Practically, this means that for instance logging statements in functions, while essentially side effects, may be permitted, as long as the function returns the same value, if given the same arguments. This is essential for correct simulation results, as the results of function calls may for instance be cached by a simulator.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *AnnotatedObject* (Section 3.10.1)

⊆ *Declaration* (Section 3.3.4)

⊆ *Function*

Direct derived classes: *ExternalFunction* (Section 3.7.5), *InternalFunction* (Section 3.7.10)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)

The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)

The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**-, **c**-, or **u**-, as those names are reserved for events.

cont **parameters** [0..*] : *FunctionParameter*

The parameters of the user-defined function.

cont **returnTypes** [1..*] : *CifType*

The return types of the user-defined function. If multiple types are defined, then the return type of the function is a tuple with nameless fields of those types.

Constraints:

- **Function.allowedReturnTypes** Component types and component definition types are not allowed in return types of functions, as components and component definitions are not meant to be used as values.

3.7.7 FunctionParameter (class)

A parameter of a user-defined function.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *FunctionParameter*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **parameter** [1] : *DiscVariable*

The discrete variable, doubling as the function parameter. Reusing the *DiscVariable* (Section 3.3.5) class makes it possible to use the same references as for discrete variables, also for function parameters.

Constraints:

- **FunctionParameter.allowedTypes** Component types and component definition types are not allowed for function parameters, as components and component definitions are not meant to be used as values.
- **FunctionParameter.noValue** The *parameter* declaration must not have a value, since the function argument should provide it.

3.7.8 FunctionStatement (abstract class)

Base class for all statements of internal user-defined functions.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *FunctionStatement*

Direct derived classes: *AssignmentFuncStatement* (Section 3.7.1), *BreakFuncStatement* (Section 3.7.2), *ContinueFuncStatement* (Section 3.7.3), *IfFuncStatement* (Section 3.7.9), *ReturnFuncStatement* (Section 3.7.11), *WhileFuncStatement* (Section 3.7.12)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

3.7.9 IfFuncStatement (class)

A conditional internal function statement.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *FunctionStatement* (Section 3.7.8)

⊆ *IfFuncStatement*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **elifs** [0..*] : *ElifFuncStatement*

The ‘else-if’ (‘elif’) alternatives. Processed in order, if the ‘guards’ evaluate to *false*.

cont **elses** [0..*] : *FunctionStatement*

The ‘else’ statements. If present, these statements are executed if no other alternative has a *true* guard.

cont **guards** [1..*] : *Expression*

The guard predicates for the ‘then’ statements.

If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between the guards.

Constraints:

- **IfFuncStatement.guardTypes** The guard predicates must have boolean types.

cont **thens** [1..*] : *FunctionStatement*

The ‘then’ statements. Executed if the ‘guards’ evaluate to *true*.

3.7.10 InternalFunction (class)

An internal user-defined function. The function is completely defined within the CIF specification.

Constraints:

- **InternalFunction.uniqueDecls** The names of all parameters and local variables in the body of the function, must be unique within that function.
- **InternalFunction.endWithReturn** All executions/evaluations of the function must end with a return statement.
- **InternalFunction.unreachable (non-fatal)** Unreachable statements are not allowed.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *AnnotatedObject* (Section 3.10.1)

⊢ *Declaration* (Section 3.3.4)

⊢ *Function* (Section 3.7.6)

⊢ *InternalFunction*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **annotations** [0..*] : *Annotation* (inherited from *AnnotatedObject*)

The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:

- Algebraic variables.
- Constants.
- Continuous variables.
- Discrete variables. But not parameters and local variables of internal user-defined functions.
- Input variables.
- Locations of automata.

attr **name** [1] : *CifIdentifier* (inherited from *Declaration*)

The name of the declaration.

Constraints:

- **Declaration.name (non-fatal)** Declaration names for non-event declarations must not start with **e**_, **c**_, or **u**_, as those names are reserved for events.

cont **parameters** [0..*] : *FunctionParameter* (inherited from *Function*)

The parameters of the user-defined function.

cont **returnTypes** [1..*] : *CifType* (inherited from *Function*)

The return types of the user-defined function. If multiple types are defined, then the return type of the function is a tuple with nameless fields of those types.

Constraints:

- **Function.allowedReturnTypes** Component types and component definition types are not allowed in return types of functions, as components and component definitions are not meant to be used as values.

cont **statements** [1..*] : *FunctionStatement*

The statements that form the body of the internal user-defined function.

cont **variables** [0..*] : *DiscVariable*

The local variable declarations of the internal user-defined function. Reusing the *DiscVariable* (Section 3.3.5) class makes it possible to use the same references as for discrete variables, also for local variables of functions.

Constraints:

- **InternalFunction.allowedVarTypes** Component types and component definition types are not allowed for local variables of functions, as components and component definitions are not meant to be used as values.
- **InternalFunction.deterministicVarInit** Local variables of functions must have at most one initial value. That is, they may not be initialized in a non-deterministic way. In other words, they are either given an explicit initial value, or they get the default value for their type. For the default values for each of the CIF types, see the *value* feature of the *DiscVariable* (Section 3.3.5) class.

3.7.11 ReturnFuncStatement (class)

A return internal function statement.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *FunctionStatement* (Section 3.7.8)
- ⊢ *ReturnFuncStatement*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **values** [1..*] : *Expression*
The return values of the return statement.

Constraints:

- **ReturnFuncStatement.types** The types of the return values must match the return types of the function. For ranged types, if a return type has a ranged type with a range, then the range of the type of the corresponding return value, must be entirely contained in the range of the return type.

3.7.12 WhileFuncStatement (class)

A while internal function statement.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *FunctionStatement* (Section 3.7.8)
- ⊢ *WhileFuncStatement*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **guards** [1..*] : *Expression*
The guard predicates for the ‘while’ statements.

If multiple predicates are given, this feature represents the logical conjunction of those predicates. Note that this represents the mathematical conjunction, and not the short-circuit conjunction binary operator. As such, there is no ordering between the guards.

Constraints:

- **WhileFuncStatement.guardTypes** The guard predicates must have boolean types.

cont **statements** [1..*] : *FunctionStatement*
The statements that form the body of the while statement.

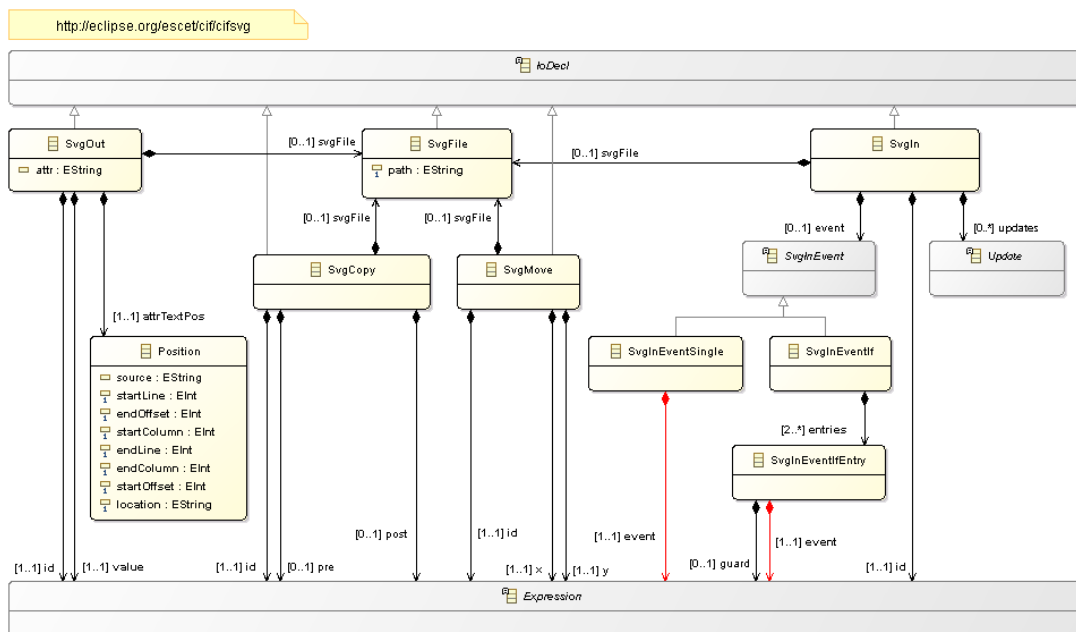


Figure 3.10: *cifssvg* package

3.8 Package cifssvg

Figure 3.10 shows the *cifssvg* package. This package contains classes that describe CIF/SVG I/O declarations, used to couple CIF models to SVG images, for SVG visualization. CIF/SVG I/O declarations fall outside of the simulation behavior/semantics of the model. The *SvgFile* (Section 3.8.2) class declares the SVG file to use, the other declarations specifies how to modify the SVG image during simulation, and how to interact with it.

Package URI <http://eclipse.org/escet/cif/cifssvg>

Namespace prefix cifssvg

Sub-packages none

3.8.1 SvgCopy (class)

SVG copy declaration. Copies a part of the SVG tree, and renames the copied elements.

Constraints:

- **SvgCopy.svgFileDefined** Either the copy declaration specifies an SVG file to use, or one of the ancestors of the copy declaration does.
- **SvgCopy.overlap (non-fatal)** The tree to copy must not be a subtree of any other tree that is copied, as the result is then dependent on the order of application of the copies.

Copying the exact same tree (same root elements) multiple times however, does not pose any problems.

- **SvgCopy.unique** The ids of the elements in copied tree, after prefixing and/or postfixing them, should be unique in the SVG image.
- **SvgCopy.prePost** A prefix value or a postfix value must be specified, or both of them, but not neither of them.
- **SvgCopy.nonRoot** Copying the root element of an SVG image's XML tree is not allowed, as copies are added as siblings of their originals, and there can only be one root element.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *IoDecl* (Section 3.2.14)
- ⊆ *SvgCopy*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **id** [1] : *Expression*
The 'id' of the SVG element that is root of the tree to copy.

Constraints:

- **SvgCopy.idType** The 'id' expression must have a string type.
- **SvgCopy.idStaticEval** The expression must be statically evaluable, after elimination of component definition/instantiation, and checking for cycles. In particular, it must not change during simulation.
- **SvgCopy.idExists** An element with the given 'id' must exist in the SVG image.
- **SvgCopy.idValidName** The 'id' must be a valid XML/SVG name.

cont **post** [0..1] : *Expression*
The text to postfix to the names of copied SVG elements.

Constraints:

- **SvgCopy.postType** The 'post' expression must have a string type.
- **SvgCopy.postStaticEval** The expression must be statically evaluable, after elimination of component definition/instantiation, and checking for cycles. In particular, it must not change during simulation.
- **SvgCopy.postValidName** The 'post' must be a valid XML/SVG name postfix.

cont **pre** [0..1] : *Expression*
The text to prefix to the names of copied SVG elements.

Constraints:

- **SvgCopy.preType** The 'pre' expression must have a string type.

- **SvgCopy.preStaticEval** The expression must be statically evaluable, after elimination of component definition/instantiation, and checking for cycles. In particular, it must not change during simulation.
- **SvgCopy.preValidName** The ‘pre’ must be a valid XML/SVG name prefix.

cont **svgFile** [0..1] : *SvgFile*

If specified, indicates the SVG image to use for the copy declaration. If not specified, the SVG image specified by the closest ancestor that specifies an SVG image is used.

3.8.2 SvgFile (class)

A declaration of the SVG file to use for the CIF/SVG declarations. If specified in a component, it applies to that component and all its children recursively, unless overridden in a deeper scope or CIF/SVG declaration. If specified in a CIF/SVG declaration, it applies only to that specific CIF/SVG declaration.

Constraints:

- **SvgFile.validSvgFile** The SVG file must exist on disk, and be a valid SVG file.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *IoDecl* (Section 3.2.14)

⊆ *SvgFile*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

attr **path** [1] : *EString*

The absolute or relative local file system path to the SVG image to use. May use both / and \ as file separators. If relative, the path is relative to the path of the CIF file.

3.8.3 SvgIn (class)

A CIF/SVG input mapping. Used to specify how clicking a certain element of the SVG image influences the traces chosen by the CIF simulator. In other words, it turns a certain element of the SVG image into an interactive SVG element.

Constraints:

- **SvgIn.svgFileDefined** Either the input mapping specifies an SVG file to use, or one of the ancestors of the input mapping does.
- **SvgIn.unique** No two input mappings may be defined for the same element ‘id’, per SVG file.

- **SvgIn.eventOrUpdate** Either the input mapping specifies which event to choose during simulation or which updates to execute during simulation.

EObject

- ⊆ *PositionObject* (Section 3.11.2)
- ⊆ *IoDecl* (Section 3.2.14)
- ⊆ *SvgIn*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **event** [0..1] : *SvgInEvent*
A specification of which event to choose during simulation, when the interactive SVG element for this input mapping is clicked.

cont **id** [1] : *Expression*
The ‘id’ of the interactive SVG element.

Constraints:

- **SvgIn.idType** The ‘id’ expression must have a string type.
- **SvgIn.idStaticEval** The expression must be statically evaluable, after elimination of component definition/instantiation, and checking for cycles. In particular, it must not change during simulation.
- **SvgIn.idExists** An element with the given ‘id’ must exist in the SVG image.
- **SvgIn.idValidName** The ‘id’ must be a valid XML/SVG name.

cont **svgFile** [0..1] : *SvgFile*
If specified, indicates the SVG image to use for the input mapping. If not specified, the SVG image specified by the closest ancestor that specifies an SVG image is used.

cont **updates** [0..*] : *Update*
A specification of which updates to execute during simulation, when the interactive SVG element for this input mapping is clicked.

Constraints:

- **SvgIn.uniqueVariables** The parts of variables that are assigned must be unique. That is, it must never be possible to assign the same part of the same variable twice. We do include the projections in the analysis, but only as far as we can statically evaluate and normalize the indices. See the ‘Edge.uniqueVariables’ constraint for the complete details.

3.8.4 SvgInEvent (abstract class)

A specification of the which event to choose during simulation, when an interactive SVG element for an input mapping is clicked.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *SvgInEvent*

Direct derived classes: *SvgInEventIf* (Section 3.8.5), *SvgInEventSingle* (Section 3.8.7)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

3.8.5 SvgInEventIf (class)

CIF/SVG input mapping event choice using an if/then/else construction.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *SvgInEvent* (Section 3.8.4)

⊢ *SvgInEventIf*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **entries** [2..*] : *SvgInEventIfEntry*

The entries of the if/then/else. The first entry is the ‘if’. The remaining entries are ‘elif’ entries. If the last entry has no ‘guard’, the entry represents and ‘else’. The entries are evaluated in the given order.

It is considered a runtime error if none of the entries can be chosen at runtime.

Constraints:

- **SvgInEventIf.else** Only the last entry may be an ‘else’, and may thus omit the ‘guard’.

3.8.6 SvgInEventIfEntry (class)

A single entry of a CIF/SVG input mapping ‘if’ event choice.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *SvgInEventIfEntry*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **event** [1] : *Expression*
The event to choose if the *guard* holds.

This feature contains *Expression* (Section 3.6.21) instances, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **SvgInEventIfEntry.event** The event reference expression must refer to an event.
- **SvgInEventIfEntry.eventInScope** The event reference must satisfy the scoping rules.

cont **guard** [0..1] : *Expression*
The guard predicates of the entry. The entry is only ‘chosen’ if the guard evaluates to *true*.
If no predicate is given, the entry acts as an emphelse, and can always be chosen (essentially a *true* guard).

Constraints:

- **SvgInEventIfEntry.guardType** The guard predicate must have a boolean type.

3.8.7 SvgInEventSingle (class)

CIF/SVG input mapping event choice that always chooses the same event.

EObject

- ⊢ *PositionObject* (Section 3.11.2)
- ⊢ *SvgInEvent* (Section 3.8.4)
- ⊢ *SvgInEventSingle*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **event** [1] : *Expression*
The event to choose.

This feature contains *Expression* (Section 3.6.21) instances, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **SvgInEventSingle.event** The event reference expression must refer to an event.
- **SvgInEventSingle.eventInScope** The event reference must satisfy the scoping rules.

3.8.8 SvgMove (class)

SVG move declaration. Moves a shape to an absolute position.

Constraints:

- **SvgMove.svgFileDefined** Either the move declaration specifies an SVG file to use, or one of the ancestors of the move declaration does.
- **SvgMove.noTransform** If an SVG element is moved, it must not also have an SVG output mapping for the ‘transform’ attribute, as that would override the move.
- **SvgMove.unique** No two move declarations may be defined for the same element ‘id’, per SVG file.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *IoDecl* (Section 3.2.14)

⊢ *SvgMove*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **id** [1] : *Expression*
The ‘id’ of the SVG element for which the shape is to be moved.

Constraints:

- **SvgMove.idType** The ‘id’ expression must have a string type.
- **SvgMove.idStaticEval** The expression must be statically evaluable, after elimination of component definition/instantiation, and checking for cycles. In particular, it must not change during simulation.
- **SvgMove.idExists** An element with the given ‘id’ must exist in the SVG image.
- **SvgMove.idValidName** The ‘id’ must be a valid XML/SVG name.

cont **svgFile** [0..1] : *SvgFile*
If specified, indicates the SVG image to use for the move declaration. If not specified, the SVG image specified by the closest ancestor that specifies an SVG image is used.

cont **x** [1] : *Expression*
The x coordinate of the upper left corner of the bounding box of the shape to move, relative to the upper left corner of the canvas, after moving.

Constraints:

- **SvgMove.xType** The ‘x’ expression must have an numeric (integer or real) type.
- **SvgMove.xStaticEval** The expression must be statically evaluable, after elimination of component definition/instantiation, and checking for cycles. In particular, it must not change during simulation.

cont **y** [1] : *Expression*

The y coordinate of the upper left corner of the bounding box of the shape to move, relative to the upper left corner of the canvas, after moving.

Constraints:

- **SvgMove.yType** The ‘y’ expression must have an numeric (integer or real) type.
- **SvgMove.yStaticEval** The expression must be statically evaluable, after elimination of component definition/instantiation, and checking for cycles. In particular, it must not change during simulation.

3.8.9 SvgOut (class)

A CIF/SVG output mapping. Used to specify how the state of the CIF model is to be used to update the SVG image, during simulation.

Constraints:

- **SvgOut.svgFileDefined** Either the output mapping specifies an SVG file to use, or one of the ancestors of the output mapping does.
- **SvgOut.unique** No two output mappings may be defined for the same element ‘id’ and attribute name, per SVG file. Similarly, no two output mappings may be defined for the same text node (taking into account that different element ids may lead to the same text node), per SVG file.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *IoDecl* (Section 3.2.14)

⊆ *SvgOut*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

attr **attr** [0..1] : *EString*

If specified, indicates the name of the attribute of which to change the value. If not specified, indicates that the text of the element is to be changed.

Constraints:

- **SvgOut.textNode** If no attribute name is specified, the SVG element must have a text node.
- **SvgOut.attrValidName** If an attribute name is specified, it must be a valid XML/SVG name.
- **SvgOut.elemNameInSvg11 (non-fatal)** If an attribute name is specified, the name of the element (not its ‘id’), must be defined in the SVG 1.1 standard.

- **SvgOut.attrNameInSvg11 (non-fatal)** If an attribute name is specified, the attribute name must be defined for the element, in the SVG 1.1 standard.
- **SvgOut.attrId** If an attribute name is specified, it must not be the `id` attribute, in any casing.
- **SvgOut.attrStyle** If an attribute name is specified, it must not be the `style` attribute, in any casing.

cont **attrTextPos** [1] : *Position*

Position information for the attribute string literal of the output mapping, or for the `text` keyword of the output mapping.

cont **id** [1] : *Expression*

The ‘id’ of the SVG element to modify.

Constraints:

- **SvgOut.idType** The ‘id’ expression must have a string type.
- **SvgOut.idStaticEval** The expression must be statically evaluable, after elimination of component definition/instantiation, and checking for cycles. In particular, it must not change during simulation.
- **SvgOut.idExists** An element with the given ‘id’ must exist in the SVG image.
- **SvgOut.idValidName** The ‘id’ must be a valid XML/SVG name.

cont **svgFile** [0..1] : *SvgFile*

If specified, indicates the SVG image to use for the output mapping. If not specified, the SVG image specified by the closest ancestor that specifies an SVG image is used.

cont **value** [1] : *Expression*

The expression to evaluate to obtain the value to use as output, to set the value of the attribute or text label.

Constraints:

- **SvgOut.valueType** Component types and component definition types are not allowed for the value, as components and component definitions are not meant to be used as values.

3.9 Package print

Figure 3.11 shows the *print* package. This package contains classes that describe print I/O declarations, used to produce textual output from CIF models. Print I/O declarations fall outside of the simulation behavior/semantics of the model. The *PrintFile* (Section 3.9.3) class declares the file (or target) to which to print the text, while the *Print* (Section 3.9.2) class specifies what to print and when to print it.

Package URI <http://eclipse.org/escet/cif/print>

Namespace prefix print

Sub-packages none

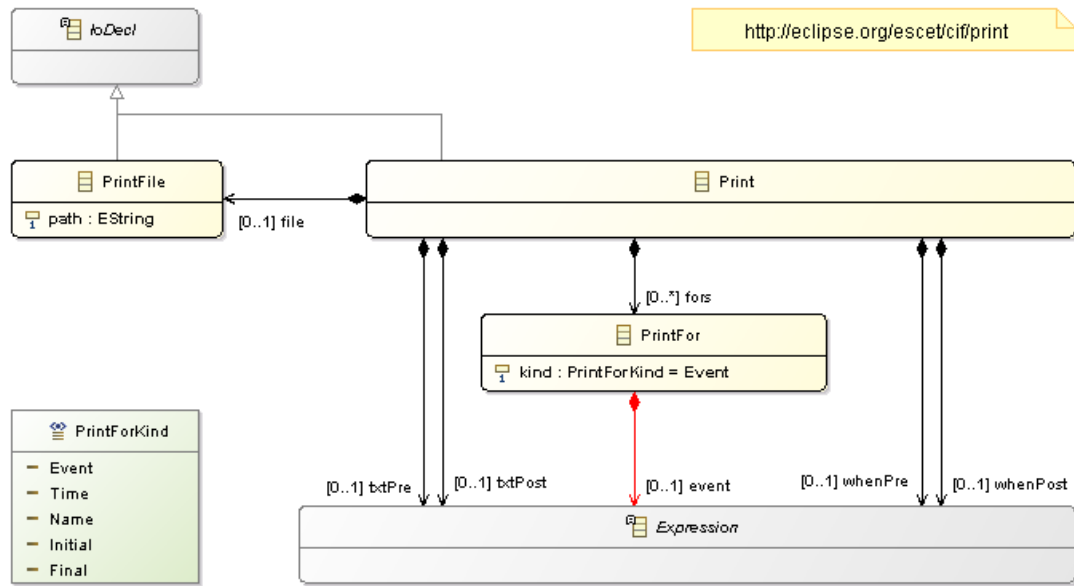


Figure 3.11: *print* package

3.9.1 PrintForKind (enumeration)

Specifies the kind of a 'for' filter of a *PrintFor* (Section 3.9.4) class.

literal **Event** (default)

Filter that includes all event transitions.

literal **Final**

Filter that includes the virtual transition after the last (final or deadlock) state.

literal **Initial**

Filter that includes the virtual transition before the first (initial) state.

literal **Name**

Filter that includes the event transitions for a specific event.

literal **Time**

Filter that includes all time transitions.

3.9.2 Print (class)

Print I/O declaration. Specifies what to print and when to print it.

Constraints:

- **Print.txtPrePost** A pre/source or post/target state text must be specified, or both of them, but not neither of them.

EObject

⊢ *PositionObject* (Section 3.11.2)

⊢ *IoDecl* (Section 3.2.14)

⊢ *Print*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

cont **file** [0..1] : *PrintFile*

The file (or target) to which to print the text.

If not specified, the file specified by the closest ancestor that specifies a *PrintFile* (Section 3.9.3) is used. If none exists, "**stdout**" is used.

cont **fors** [0..*] : *PrintFor*

The ‘for’ filters for this print declaration. Only transitions for which at least one of the filters matches, lead to text being printed. If no filters are specified, it defaults to **initial**, **event**, **time**.

Constraints:

- **Print.duplFor (non-fatal)** The ‘for’ filters should not contain duplicates (be over-specified). That is, specifying the same kind multiple times should be avoided (except *PrintForKind.Name* (Section 3.9.1)). Furthermore, specifying the same event multiple times should be avoided. Finally, specifying *PrintForKind.Event* (Section 3.9.1) (all events) and *PrintForKind.Name* (Section 3.9.1) (specific event) should be avoided.

cont **txtPost** [0..1] : *Expression*

If specified, indicates the text to print for the post/target state.

Evaluation result is converted to a textual representation. String typed results are a special case, as no double quotes are added, and escaping is not performed.

Constraints:

- **Print.txtPostType (non-fatal)** The type of a print declaration post text, if specified, must not have a component or component definition type.

cont **txtPre** [0..1] : *Expression*

If specified, indicates the text to print for the pre/source state.

Evaluation result is converted to a textual representation. String typed results are a special case, as no double quotes are added, and escaping is not performed.

Constraints:

- **Print.txtPreType (non-fatal)** The type of a print declaration pre text, if specified, must not have a component or component definition type.

cont **whenPost** [0..1] : *Expression*

If specified, filters post/target states. Text is only printed if the predicate holds for a post/target state.

Constraints:

- **Print.whenPostType** The ‘when’ ‘post’ predicate must have a boolean type.

cont **whenPre** [0..1] : *Expression*

If specified, filters pre/source states. Text is only printed if the predicate holds for a pre/source state.

Constraints:

- **Print.whenPreType** The ‘when’ ‘pre’ predicate must have a boolean type.

3.9.3 PrintFile (class)

The file (or target) to which to print text. Applies to the scope in which it is specified, and all child scopes recursively, unless overridden in a deeper scope.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *IoDecl* (Section 3.2.14)

⊆ *PrintFile*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)

Optional position information.

attr **path** [1] : *EString*

The path of file (or target). If it refers to a file, it must be an absolute or relative local file system path to the file. May use both / and \ as file separators. If relative, the path is relative to the path of the CIF file. Special values, such as `":stdout"`, are supported as well.

3.9.4 PrintFor (class)

A ‘for’ filter of a *Print* (Section 3.9.2). Filters transitions to keep only those that match the ‘for’ filter.

Constraints:

- **PrintFor.eventSpecified** The ‘event’ must be specified if and only if the ‘kind’ is *PrintForKind.Name* (Section 3.9.1).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *PrintFor*

Direct derived classes: none

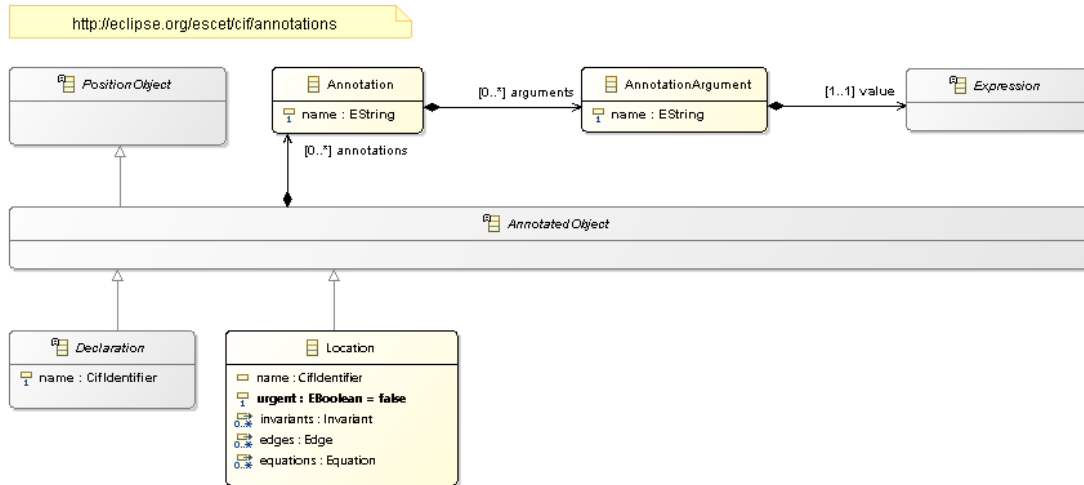


Figure 3.12: *annotations* package

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **event** [0..1] : *Expression*
The event to include, for this ‘for’ filter.

This feature contains *Expression* (Section 3.6.21) instances, to allow for wrapping expressions to be used. See also Section 3.1.

Constraints:

- **PrintFor.event** The event reference expression must refer to an event.
- **PrintFor.eventInScope** The event reference must satisfy the scoping rules.

attr **kind** [1] : *PrintForKind*
The kind of the ‘for’ filter.

3.10 Package annotations

Figure 3.12 shows the *annotations* package. The main class is the *Annotation* (Section 3.10.2) class. It represents a single annotations on a single object. The *AnnotationArgument* (Section 3.10.3) class represents a single arguments of an annotation. And the *AnnotatedObject* (Section 3.10.1) class is the base class for all objects that can be annotated.

Package URI <http://eclipse.org/escet/cif/annotations>

Namespace prefix annotations

Sub-packages none

3.10.1 AnnotatedObject (abstract class)

Base class for all objects that can be annotated.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *AnnotatedObject*

Direct derived classes: *Declaration* (Section 3.3.4), *Location* (Section 3.4.10)

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **annotations** [0..*] : *Annotation*
The annotations of the annotated object.

Constraints:

- **AnnotatedObject.onlyForSupportedObjects** Currently only the following objects support annotations:
 - Algebraic variables.
 - Constants.
 - Continuous variables.
 - Discrete variables. But not parameters and local variables of internal user-defined functions.
 - Input variables.
 - Locations of automata.

3.10.2 Annotation (class)

An annotation.

Constraints:

- **Annotation.annotationSpecificErrors** The annotation and its arguments must satisfy any additional constraints imposed on the annotation by the registered annotation provider.
- **Annotation.annotationSpecificWarnings (non-fatal)** The annotation and its arguments must satisfy any additional constraints imposed on the annotation by the registered annotation provider.

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *Annotation*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

cont **arguments** [0..*] : *AnnotationArgument*
The arguments of the annotation.

Constraints:

- **Annotation.uniqueArguments** The names of each of the arguments must be unique within the annotation.

attr **name** [1] : *EString*
The name of the annotation.

Constraints:

- **Annotation.validName** The name must be a CIF identifier, or multiple CIF identifiers joined together using colons (:). For instance, **name** or **some:name**.
- **Annotation.registeredName (non-fatal)** The annotation name must be a registered annotation name, within the environment where the CIF model is used.

3.10.3 AnnotationArgument (class)

An argument of an *Annotation* (Section 3.10.2).

EObject

⊆ *PositionObject* (Section 3.11.2)

⊆ *AnnotationArgument*

Direct derived classes: none

cont **position** [0..1] : *Position* (inherited from *PositionObject*)
Optional position information.

attr **name** [1] : *EString*
The name of the annotation argument.

Constraints:

- **AnnotationArgument.validName** The name must be a CIF identifier, or multiple CIF identifiers joined together using periods (.). For instance, **arg** or **some.arg**.

cont **value** [1] : *Expression*
The value of the annotation argument.

3.11 Package position

Figure 3.13 shows the *position* package.

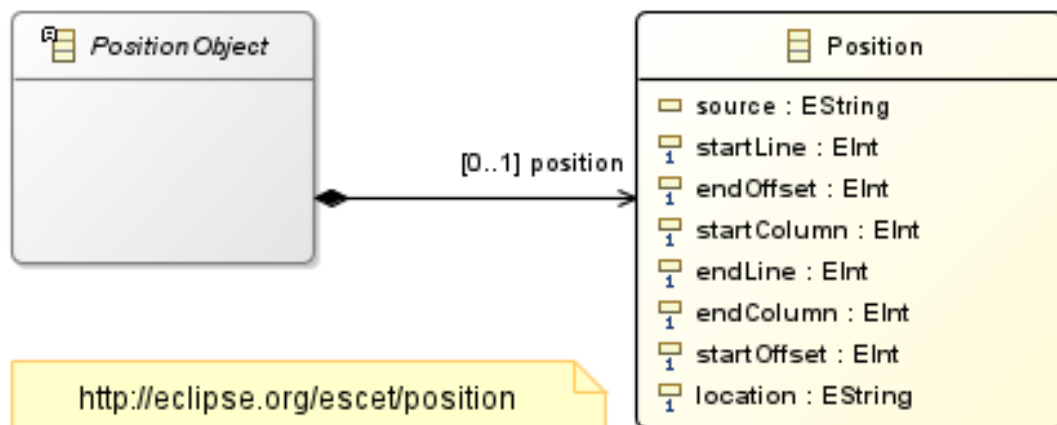


Figure 3.13: *position* package

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 3.11.1) class represents actual position information. The abstract *PositionObject* (Section 3.11.2) 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`

3.11.1 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.

3.11.2 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

Distributions

This chapter gives an overview of the stochastic distributions that are available in CIF, as well as their properties.

Tables 4.1, 4.2, and 4.3 respectively list the constant, discrete, and continuous distributions present in CIF. Each of the tables lists the name of the distribution in their first column, and the parameters and sample type in the second column. In the third column, they list the name of the distribution and its parameters as used in a book by Law & Kelton [1], except for the *StdLibFunction.Constant* (Section 3.6.2) distributions. Those distributions return the provided parameter value as sampled value each time. Their main use is for testing and debugging. The fourth, fifth, and sixth column list the mean, variance, and range of the distributions respectively.

For the *Weibull* function, $\Gamma(z)$ is the gamma function, defined as $\Gamma(z) = \int_0^\infty t^{(z-1)} \cdot e^{-t} dt$ for all real numbers $z > 0$.

Name	Type signature	Book	Mean	Variance	Range
<i>Constant distributions returning the specified value.</i>					
Constant	bool (bool b)	-	b	0	$\{true, false\}$
Constant	int (int i)	-	i	0	$(-\infty, \infty)$
Constant	real (real r)	-	r	0	$(-\infty, \infty)$

Table 4.1: Constant distributions and their properties.

Name	Type signature	Book	Mean	Variance	Range
<i>Bernoulli distribution with chance $p \in [0, 1]$ for true.</i>					
Bernoulli	bool (real p)	Bernoulli(p)	p	$p(1 - p)$	$\{true, false\}$
<i>Binomial distribution with $t > 0$ experiments with chance $p \in [0, 1]$.</i>					
Binomial	int (real p , int t)	bin(t , p)	$t \cdot p$	$t \cdot p(1 - p)$	$\{0, 1, 2, \dots, t\}$
<i>Geometric distribution, number of failed Bernoulli(p) experiments with chance $p \in (0, 1]$ before first success.</i>					
Geometric	int (real p)	geom(p)	$\frac{1-p}{p}$	$\frac{1-p}{p^2}$	$\{0, 1, 2, \dots\}$
<i>Poisson distribution with rate $r > 0$.</i>					
Poisson	int (real r)	P(r)	r	r	$\{0, 1, 2, \dots\}$
<i>Discrete uniform distribution with $a < b$.</i>					
Uniform	int (int a , b)	DU(a , $b - 1$)	$\frac{a+b-1}{2}$	$\frac{(b-a)^2-1}{12}$	$\{a, a + 1, a + 2, \dots, b - 1\}$

Table 4.2: Discrete distributions and their properties.

Name	Type signature	Book	Mean	Variance	Range
<i>Beta distribution with shape parameters $a > 0$ and $b > 0$.</i>					
Beta	real (real a , b)	B(a , b)	$\frac{a}{a+b}$	$\frac{a \cdot b}{(a+b)^2 \cdot (a+b+1)}$	$[0, 1]$
<i>Erlang distribution with parameter $m > 0$ and scale parameter $b > 0$, also known as the Gamma(m, b) distribution.</i>					
Erlang	real (int m , real b)	m -Erlang(b)	$m \cdot b$	$m \cdot b^2$	$[0, \infty)$
<i>Negative exponential distribution with scale parameter $b > 0$.</i>					
Exponential	real (real b)	expo(b)	b	b^2	$[0, \infty)$
<i>Gamma distribution with shape parameter $a > 0$ and scale parameter $b > 0$.</i>					
Gamma	real (real a , b)	Gamma(a , b)	$a \cdot b$	$a \cdot b^2$	$[0, \infty)$
<i>Lognormal distribution with $b > 0$.</i>					
LogNormal	real (real a , b)	LN(a , b)	$e^{a+\frac{b}{2}}$	$e^{2a+b}(e^b - 1)$	$[0, \infty)$
<i>Normal distribution with $b > 0$.</i>					
Normal	real (real a , b)	N(a , b)	a	b	$(-\infty, \infty)$
<i>Random distribution, shorthand for the continuous uniform distribution from 0 to 1.</i>					
Random	real ()	U(0, 1)	$\frac{1}{2}$	$\frac{1}{12}$	$[0, 1)$
<i>Triangle distribution from a to c with the top at b, $a < b < c$.</i>					
Triangle	real (real a , b , c)	triang(a , c , b)	$\frac{a+b+c}{3}$	$\frac{a^2+b^2+c^2-a \cdot b-a \cdot c-b \cdot c}{18}$	$[a, c]$
<i>Continuous uniform distribution from a to b, with $a < b$.</i>					
Uniform	real (real a , b)	U(a , b)	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$	$[a, b)$
<i>Weibull distribution with shape parameter $a > 0$ and scale parameter $b > 0$.</i>					
Weibull	real (real a , b)	Weibull(a , b)	$\frac{b}{a} \cdot \Gamma(\frac{1}{a})$	$\frac{b^2}{a} \cdot (2 \cdot \Gamma(\frac{2}{a}) - \frac{\Gamma(\frac{1}{a})^2}{a})$	$[0, \infty)$

Table 4.3: Continuous distributions and their properties.

Chapter 5

Legal

The material in this documentation is Copyright (c) 2010, 2024 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] Averill M. Law and David Kelton. *Simulation modeling and analysis*. McGraw-Hill, New York, second edition, 1991.
- [2] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. Addison-Wesley, 2009.
- [3] Contributors to the Eclipse Foundation. Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET). <https://eclipse.dev/escet>.