Constraint Modeling Language (CML) User Guide

Edition 3.0
Spring ‘26

What Is Constraint Modeling Language?

Constraint Modeling Language (CML) is a domain-specific language that defines models for complex systems. For product configuration, constraint models describe real-world entities and their relationships to each other. Constraint models enforce business logic declaratively, without the need for extensive code in a general-purpose programming language. The constraint engine compiles CML code into a constraint model and uses the model to construct a product configuration that complies with the specified constraints.

To build a constraint model in CML, use this basic workflow:

- Create global properties and settings which are header-level declarations in CML that define the foundational, fixed values for the entire constraint model. They are crucial for setting up the core configuration environment and ensuring reusability across the model.
- Create variables to define the properties or characteristics of a type. Variables can hold different kinds of data, such as strings, numbers, or lists, and can be calculated from other variables and values. In Revenue Cloud, variables represent product fields, product attributes, and sometimes context tags. See Create a Context Definition in Salesforce Help.
- Define types, which represent entities or objects in the model. Types are the building blocks of CML. They're similar to classes in object-oriented programming. In Revenue Cloud, types represent standalone products, bundles, product components, and product classes.
- Define relationships that describe how different types are associated with each other. In Revenue Cloud, relationships represent the product structure in a bundle. For example, the root product has a relationship with its components.
- Apply constraints to define logical restrictions, and enforce rules and conditions on types, variables, and relationships.

Note: To define a constraint for a child product in a bundle, you must include the entire bundle in the constraint model. For example, if you define a constraint for a laptop, and the laptop is a child product in the Laptop Pro Bundle, you must include the Laptop Pro Bundle in the constraint model for the constraint on the laptop to run.

See Constraint Model Example: Modeling a Generator Set and CML Core Concepts.

Working with CML in Salesforce

Use CML to create and edit constraint models with Constraint Rules Engine in Product Configurator in Salesforce. For more information on working with CML in Salesforce, see these Salesforce Help articles.

- Use Constraint Builder With Constraint Rules Engine
- Define Constraints and Rules with the Visual Builder
- Use Code to Define Constraints and Rules in the CML Editor

Constraint Model Example: Modeling a Generator Set

The Constraint Model for a Generator Set examples use CML to define a technical power configuration, illustrating concepts such as calculated variables, enforcement of external standards, and component selection based on requirements. The examples correspond to the CML core concepts linked here. See the Generator Set examples for code samples that use the core concepts.

- Global Properties and Settings: `VOLTAGE_REGEX` is a global constant that defines a fixed regular expression pattern used for validation or parsing throughout the model.
- Types: `GeneratorSet` is the root type that represents the main entity. `GeneralModel` represents a related component type.
- Variables: The `GeneratorSet` type defines variables like `requiredKW` (the user's power requirement), `Voltage`, and calculated variables like `surgeLoadKW` and `Voltage3` (derived from parsing the `Voltage` string).
- Relationships: The `GeneralModels` relation connects the `GeneratorSet` type to its possible configurations (`GeneralModel`).
- Constraints: Constraints enforce critical business rules and safety standards, such as ensuring the selected generator model's power meets the required threshold, or restricting configuration options (like `Voltage`) based on the specified compliance standards (`Listing-UL 2200`).

CML Core Concepts

See these topics for information on each core concept and the ways they work together.

- Global Properties and Settings
- Variables
- Types
- Relationships
- Constraints

Note: CML supports single-line code comments with `//` and block comments with `/* */`.

Global Properties and Settings

Header-level declarations define the global properties and settings for a model, including constants, properties, and external values that set up the foundation of the CML code. Use these declarations to create reusable components and configuration settings that you can reference throughout the model.

Global Constants

Use global constants to define values that remain fixed throughout the model. These constants can be numeric values, strings, lists, or other supported data types. Use constants to create standardized settings or options that you can reference multiple times. See Example 1: Use Regex Global Variable.

In the example, `MAX_COUNT` is a global constant that is hard-coded to `100`:
define `MAX_COUNT 100`

Regex (regular expressions) can be used to define global constants. The generalized abstract syntax structure for regex expressions is:
`define <CONSTANT_NAME> "^<REGEX_PATTERN_STRING>$"`

Regex Pattern Components

This table lists regex components and their details.

Regex Component	Description	Generalization
`^` and `$`	Anchors that ensure the pattern matches the entire string, from the beginning (`^`) to the end (`$`).	Ensures strict adherence to the required format.
`()`	Capturing Groups used to isolate portions of the matched string. You can reference the captured parts later by using `$1`, `$2`, and so on, in functions such as `regexpreplace`. See String Variable Functions and Operators.	Allows extraction of specific data fields from a string.

`+`	Character Class and Quantifier that matches one or more (`+`) digits or characters.	Defines the permitted characters and minimum occurrences for the data fields.
`/`	Literal Character matching the forward slash separator present in the input data.	Matches fixed delimiters in the input string.

In the example, `VOLTAGE_REGEX` is a global constant that defines a fixed regular expression pattern used for validation or parsing throughout the model:

```
define VOLTAGE_REGEX "^([0-9]+)/([0-9]+)$"
```

For more on the usage of global properties, see External Variables.

Variables

Variables are the properties or characteristics defined within a type. Variables can hold different types of data, such as strings, numbers, or lists, and can be calculated from other values.

Variable Domains and Domain Restrictions

A variable can have a fixed domain that defines a set of permitted values. You can specify the domain as:

- A list of discrete values
- A continuous range
- A combination of ranges and discrete values

For more information, see domainComputation in Variable Annotations.

Example

This example defines a `SwitchgearBay` type with three variables, each having a fixed domain:

- `BayType` can be one of the specified string values.
- `Bay_Number` can be any integer between 1 and 9 (inclusive).
- `Total_Power_Required_kW` can be any integer between 1 and 100000 (inclusive).

```
type SwitchgearBay {
    string BayType = ["load", "lv", "mv"];
    int Bay_Number = [1..9];
    int Total_Power_Required_kW = [1..100000];
}
```

Variable Data Types

Variables support multiple data types including boolean, date, decimal, and so on. Variables without a domain definition may remain unbound, leading to errors.

Data Type	Description	Defaulting Example
boolean	Only `true`, `false`, or `null` can be assigned as a value.	`@(defaultValue="true") boolean isActive;`
date	A value that indicates a particular day, the same as local date in Java.	`date shipDate = ["2023-01-01", "2023-12-31"];` If there's no `defaultValue` specified, the variable defaults to the first value in the domain. See the Constraints using Format Specifiers (%s, %d) and Dates example.
double(n)	A 64-bit number that includes a decimal point, the same as double in Java.	`double(2) percentage = [0.00..100.00];` If there's no `defaultValue` specified, the variable defaults to the first value in the domain.
decimal(n)	A fixed-point numeric value with n decimal places.	`decimal(2) TaxRate = 0.08;`
int	Integer. A 32-bit number that doesn't include a decimal point, the same as int in Java.	`@(defaultValue = "5") int defaultQty = [1..10];` If there's no `defaultValue` specified, the variable defaults to the first value in the domain.
string	Any set of characters surrounded by double quotes (“)	`@(defaultValue = "Red") string color = ["Red", "Green", "Blue"];` If there's no `defaultValue` specified, the attribute defaults to the first value in the domain.
string[]	Used in multi-select picklists for the user to select more than	`@(defaultValue = '["Red", "Green"]') string[]`

	one item from multiple options. For example, if a user selects “Red”, “Green”, and “Blue” values in a color picker, this variable holds those selected values.	`selectedColors;` If there’s no `defaultValue` specified, the attribute picklist defaults to the first value in the domain.

Variable Functions

CML variable functions are fundamental tools used to perform both aggregation (summarizing data from related components) and complex mathematical calculations on attribute values (variables) within a configuration model. These functions are crucial for enforcing dimensional validity and calculating derived attributes.

You can use functions such as `sum()`, `min()`, `max()`, `count()`, and `total()` to calculate values from all variables with the same name in the descendants of the current type.

Aggregate Functions (Data Summarization)

Aggregate functions are used primarily as Port Attributes within a relation to calculate values across multiple instances of a component type (descendants).

Function	Purpose	CML Keyword [Source]
`sum() / total()`	Calculates the sum of a specific numeric attribute across all instances in a relationship. `total()` is an alias for `sum()`.	`sum, total`
`count()`	Counts the number of component instances within a relationship that match a specific logical condition.	`count`
`max() / min()`	Returns the maximum or minimum value of an attribute found across all instances in a relationship.	`max, min`

Mathematical Functions (Numerical Derivation)

Mathematical functions and operators are used to calculate derived values based on arithmetic relationships between variables.

Function/Operator	Purpose	CML Keyword/Operator [Source]
Arithmetic Operators	Perform standard arithmetic: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), modulo (`%` or `mod`), and power (`^`).	`+, -, *, /, %, ^`
`ceil()`	Returns the smallest integer greater than or equal to the argument (rounds up).	`ceil`

Usage Example

```
surgeLoadKW == requiredKW * 1.25);
accessoryWeight = sum(weight_kg);
accessoryCount = count(weight_kg > 0);
ceil(totalItems / itemsPerCrate)
```

See the full example in Arithmetic Calculations and Functions.

String Variable Functions and Operators

CML provides string manipulation and conversion functions, and string comparison and validation operators. The functions can be used to modify strings, extract information, or convert strings to numeric types. The operators can be used to validate string values against sets or regular expressions.

Function	Purpose	Syntax, Examples, and Additional Details
`strlen()`	Returns the length of the string as an integer.	`strlen(inputString)` `strlen("Hello"` returns 5.
`substr()`	Returns a substring from the input string.	`substr(inputString, startIndex)` or `substr(inputString, startIndex, endIndex)`

		The index starts with 0. The character at `startIndex` is included, and the character at `endIndex` is excluded.
`strconcat()`	Concatenates multiple strings using a specified separator.	`strconcat(separator, stringArgsToConcatenate)` `strconcat("-", "a", "b")` results in `"a-b"`.
`join()`	An aggregate function that concatenates string values across related components, typically using a separator. The separator is a comma by default.	`names = join(name)`
`trim()`	Returns the string after removing any leading or trailing spaces.	`trim(strToTrim)`
`strsplit()`	Splits a string into an array of strings around a given separator.	`strsplit(stringToSplit, separator)`
`strcontain()`	Returns a boolean value (true or false) to specify whether the input string contains the specified search string.	`strcontain(inputString, searchString)`
`strshare()`	Splits two strings using a delimiter (comma by default) and returns true if the resulting lists have any elements in common.	`strshare(string1, string2, delimiter)`
`strformat()`	Returns a formatted string based on parameters and format specifiers.	`strformat("%d person", quantity)`

		Specifiers include `%d` for integer, `%s` for string, `%.2f` for decimal/float, and `%b` for boolean.
`strtoint()`	Converts a string to an integer.	`strtoint(inputString, defaultValue)` If the string can't be parsed as an integer (for example, it contains text or a decimal point), the provided defaultValue is returned.
`strtofloat()`	Converts a string to a decimal (floating point) value.	`strtofloat(inputString, defaultValue)` If the conversion fails, the provided `defaultValue` is returned. The resulting decimal attribute must be declared with the intended precision. For example, `decimal(3)`.
`regexpreplace()`	Take an input string and a defined Regular Expression (regex). Replace the entire input string with a specific captured group from the regex match.	`regexpreplace(InputString, RegexPattern, ReplacementGroup)` This function is commonly used to derive numeric data from complex, formatted strings before they are converted into integers or decimals.
`get(index, array)`	Used to retrieve an element at a specified index (starting at 0) from an array or list generated by a function like strsplit.	`string splitStr1 = get(0, strsplit("hello constraint engine", " ")); // returns hello` **Using Get with Table Constraint** If your table constraint output for three columns is stored in a string[] variable named picklistValues, you would access the first row's attributes like this (assuming the columns are stored in a predictable sequence):

		`// Retrieve the first element (Row 0, Column 0)`
		`string value0_0 = get(0, picklistValues);`
		
		`// Retrieve the second element (Row 0, Column 1)`

String Comparison and Validation Operators

The `in operator` checks if the value of a string attribute is present within a defined set or array of strings.

Example

```
color in ["red", "blue"]
```

Variable Annotations

In this example, the `gc_runningKw` variable is annotated to indicate that it's not configurable and has a default value of `0.00`:

```
@(configurable = false, defaultValue = 0.00)
decimal(2) gc_runningKw;
```

Note: This example requires an enclosing type.

You can annotate variables with properties.

Property	Values	Description
allowOverride	true, false	Allows the engine to recalculate a value even if a value was already received from core. This annotation helps save on performance by allowing early calculation.
configurable	true, false	Indicates whether the variable is configurable. If the value is false, the configuration engine doesn't assign a value to the variable.
defaultValue	literal	Indicates the default value for the variable. **Note**: If there's no `defaultValue` specified, the

		attribute choice for an item in Product Configurator defaults to the first value in the range.
domainComputation	true, false	Indicates whether the variable has a fixed set of values (a fixed domain). If the value is true, the values update when the configuration is changed. If the value is false, the values are fixed and don't update.
nullAssignable	true, false	Sets an initial value for the calculated variable if the expression value can't be calculated.
productGroup	integer	Used to represent the group cardinality for relationships under a type, either for specified relationships, or for all relationships (using`*`). **Note**: We recommend using maxInstanceQty and minInstanceQty type annotations instead of productGroup. See Type Annotations.
relatedAttributes	string value	relatedAttributes is required to reset the domain to the original domain for domain computation.

sequence	integer	Indicates the sequence in which variables are configured.
		
		The constraint engine assigns the values based on the sequence, with the lowest in the sequence assigned first.
		
		Example
		`type Desktop {`
		`@(defaultValue = "1080p Built-in Display", sequence=1)`
		`string Display = ["1080p Built-in Display", "4k Built-in Display", "2k Built-in Display"];`
		
		`@(defaultValue = "15 Inch", sequence=2)`
		`string Display_Size = ["15 Inch", "24 Inch", "13 Inch", "27 Inch"];`
		`}`

setDefault	true, false	Sets the variable status to default.
source	string value	Data source defined in the model.
sourceAttribute	Variable name in string	Sets the domain of the current variable to be the domain of the source variable.
strategy	descending, ascending, string	Defines the strategy to configure the variable. • If the strategy is ascending, the engine tries values from low to high. • If the strategy is descending, the engine tries values from high to low.

tagName	string value	tagName can be a comma-delimited string. Maps the tag from context definitions to the type attribute in the model.
		
		Example
		`type Quip {`
		`string SubscriptionType = ["Business", "Enterprise"];`
		
		`@(tagName = "ItemSubtotal")`
		`decimal ItemSubtotal1;`
		`constraint(ItemSubtotal1 > 300 -> SubscriptionType=="Enterprise", "If Subtotal is greater than $300 the subscriptionType should be Enterprise");`
		`}`
		
		To create an external variable linked to a Sales Transaction header, use the tagName annotation with the contextPath annotation to reference context tags on SalesTransactionItem within a type. See contextPath in External Variable Annotations.

External Variables

External variables are global CML variables that are defined within a virtual CML type. See virtual in Type Annotations. The values for external variables are usually set by the environment that launches the constraint solver engine. Use external variables to import runtime data from the context header (such as Quote or Sales Transaction) into the configuration model, with the `contextPath` annotation to denote header fields, or with tagName annotation to denote lineItem fields. See External Variable Annotations.

If the external variable isn't mapped to any external data source, it must have a default value. Otherwise, it may remain unbound and cause errors.

Basic Declaration Syntax

```
extern int MAX_VALUE = 9999;
extern decimal(2) threshold = 1.5;
```

Example: Using External Variables with Context Path Annotation

In this example, the constraint engine needs access to the quote header (Sales Transaction) field, which defines the shipping location to enforce region-specific compliance requirements. The `contextPath` annotation is used to map the field (`SalesTransaction.ShippingCountry`) to an external CML variable (`ShippingCountry`).

Note: The CML variable name can be different from the context path value.

```
// External variable declaration with context path annotation
@(contextPath = "SalesTransaction.ShippingCountry")
extern string ShippingCountry; // ShippingCountry Value is
pulled from the Quote/Order header
```

See the full example in Using ContextPath and tagName annotations.

External Variable Annotations

Here are the details of external variable annotations.

Annotation	Possible Value	Description
contextPath	"SalesTransaction.<ST_FIELD>", where the sales transaction field is pulled directly from the context definition.	References sales transaction values directly from their context definition, such as account name, sales transaction total, or address. The

		contextPath annotation can only be used for header fields. To create a variable linked to a SalesTransactionItem, use the tagName annotation to reference context tags on SalesTransactionItem within a type. See tagName in Variable Annotations.

Types

In CML, you define types to represent entities or objects in the model. Types are the foundational building blocks of CML. A type encapsulates the property, relationships, constraint, and rules for the entity. A type is similar to a class in object-oriented programming.

You can define relationships that represent associations between different types. See Relationships.

Generic Structure of a Type

This table explains the generic structure of types with examples.

Element	Purpose	Example

Declaration	Defines the entity name, optionally preceded by annotations and optionally followed by inheritance, if applicable.	`type Product {}`
		
		`Or optionally: @annotation_name("annotation parameters") type Product: BaseProduct {}`

Variables (Attributes)	Defines the properties or characteristics of the entity, including data type and domain.	`int requiredKW = [101..10000]; string color = ["Red", "Blue"];`

Relations	Defines one-to-many associations with other types, specifying cardinality (the quantity range) and optionally, the configuration order.	`relation items : LineItem[1..10] {}`
Constraints and Rules	Enforces business logic and restrictions that must be satisfied by the entity's variables and relationships.	`constraint(condition);` `require(condition, items [type]);`

Example: Basic Type Declaration with Variables

This example shows the declaration of the main `GeneratorSet` type. It defines several core attributes (variables) that characterize the product.

```
type GeneratorSet{
   // Declaration only (inherits LineItem properties)

   // Attributes with explicit domains
   int requiredKW = [101..10000];
   string Voltage = ["220/380", "240/416", "255/440", "277/480",
"347/600", "2400/4160", "7200/12470", "7621/13200",
"7976/13800"];
   string DutyRating = ["Prime Power (PRP)", "Continuous Power
(COP)", "Data Center Continuous (DCC)", "Emergency Standby Power
(ESP)"];
}
```

Type Hierarchies

CML supports inheritance and overriding, which allow you to create hierarchies between types.

How Hierarchies Function

- Inheritance: This mechanism enables a specialized child type to automatically share common variables (attributes) and relationships defined in its parent type. By inheriting from a base type, child types don’t need to redefine shared properties, which ensures consistency across the model.
- Overriding: While child types inherit the structure of the parent, they can override or specialize those properties. For example, a parent type might define a variable with a

broad range of possible values, while a child type overrides that variable with a fixed value specific to that individual product.

Practical Examples of Hierarchy

1. Simple Product Extension: A `BaseProduct` might define an `id` and `name`. A `PhysicalProduct` can then inherit from `BaseProduct` to gain those fields while adding its own unique characteristics like `weight` or `color`.
2. Multi-Level Nesting: Hierarchies can extend through multiple layers. For instance, a `Room` type can be the parent to a `Bedroom` type, and a `MasterBedroom` can further inherit from the `Bedroom` type, carrying all properties down the chain.
3. Abstract Base Models: In complex configurations like a `GeneratorSet`, a parent type like `GeneralModel` defines the necessary attributes (such as `powerKW` and `dB`), while specific child types like `GeneralModel900` or `GeneralModel1200` inherit those attributes and override them with their specific ratings.

Core Benefits

By establishing these hierarchies, constraint models become more modular and efficient. You can create header-level declarations and base types to serve as a foundation for the entire model, allowing you to reference reusable components multiple times rather than writing redundant code for every product variation. This structural organization allows the constraint engine to effectively enforce business logic and validate configurations across all related types.

Example 1: Simple Product Extension

In this pattern, a specialized type inherits from a broader base type. In the provided generator set model, the `GeneratorSet` type inherits from `LineItem`, gaining any properties defined at the line-item level while adding its own specific configuration fields like `requiredKW` and `Voltage`.

```
// The ultimate base type in the system
type LineItem;
// Child type extending LineItem with generator-specific
attributes
type GeneratorSet : LineItem {
   int requiredKW = [101..10000];
   string Voltage = ["220/380", "240/416", "255/440"];

   string DutyRating;
}
```

Example 2: Multi-Level Nesting

CML supports hierarchies with multiple layers of depth. In this example, properties flow from the top-level `LineItem` down to the `GeneratorSet`, and finally to a highly specialized `EmergencyGenerator`. Each level inherits all attributes from the levels above it.

```
type LineItem;
// Level 2: Adds basic generator capacity attributes
type GeneratorSet : LineItem {
   int requiredKW = [101..10000];
   decimal(2) surgeLoadKW = requiredKW * 1.25; // Calculation
shared down the chain
}
// Level 3: Specialized version inheriting everything from
GeneratorSet and LineItem
type EmergencyGenerator : GeneratorSet {
   // Automatically inherits requiredKW and surgeLoadKW
   string DutyRating = "Emergency Standby Power (ESP)"; //
Specific fixed rating
}
```

Example 3: Abstract Base Models (Polymorphism & Overriding)

This pattern uses an abstract type as a structural blueprint. Specific components (the "General Models") inherit from this blueprint and override its generic attributes with fixed, real-world ratings.

```
// Base model acting as a blueprint for all engine options
type GeneralModel{
   int powerKW = [100..2000]; // Broad domain
   int dB = [60..100];
}

// Specific product type that overrides parent domains with
fixed values
type GeneralModel900 : GeneralModel {
   int powerKW = 900; // Overrides broad range with exact value
   int dB = 78;
}

// Another specialized model with different fixed properties
type GeneralModel1500 : GeneralModel {
```

```
    int powerKW = 1500;
    int dB = 83;
}
```

Type Annotations

You can annotate types to add information. Type annotations are metadata applied to a type declaration to provide instructions to the constraint engine regarding how instances of that type should be handled, instantiated, or used in the configuration structure.

Annotation	Possible Values	Description
virtual	true, false	If `true`, specifies whether the indicated type refers to the transaction header (such as Quote or Order) or to a logical container (sub group of the Quote or Order). `false` is the default behavior for types and doesn't need to be explicitly specified.
groupBy	Variable name	Used with `virtual = true`, the `groupBy` annotation organizes child products—the individual instances populating a relationship—into virtual containers based on a shared attribute value. See Relationships and the Grouping Generators by Voltage example.
maxInstanceQty	Integer	Specifies the maximum cardinality for a component in a group. See Group Type.
minInstanceQty	Integer	Specifies the minimum cardinality for a component in a group. See Group Type.
source	String	Specifies the data source defined in the model.
split	true, false, none	Specifies whether the type should be split or not. • If `split=true`, there can be multiple instances of the type, and the quantity of each instance is always 1. • If `split=false`, there is only one instance in the relationship. If the user adds more instances, the engine adds more quantity to the existing instance.

		• If `split=none` (the default), there are multiple instances of the same type in the relationship, with different quantities. **Note**: The `split=true` annotation isn't supported for child products within a dynamic bundle.
sharingcount	Integer	Specifies the maximum number of times a single instance of a specific type can be shared or reused across different relationships within the configuration model. This annotation is used in conjunction with the `@(split=true)` annotation. When a type is marked for splitting, the constraint engine can process multiple instances in parallel to improve performance. The `sharingCount` tells the engine exactly how many times it can "split" or reuse that instance to satisfy the configuration requirements without generating entirely new, unique instances. It's a critical tool for managing large-scale configurations (for example, models with over 1,000 components). By setting a sharing limit, you reduce the number of variables the engine must instantiate, which helps prevent performance degradation and system timeouts. The `sharingCount` annotation works with the `@(sharing=true)` annotation applied to Relations. The relation annotation enables the general capability to share components across instances, while the `sharingCount` on the child type sets the numerical limit for that behavior. See Relationship Annotations and the Sharing Accessories in a Generator Set example.

Example 1: Enforcing Split Instances (@split = true)

In this example, the `@split = true` annotation is used on component types (such as `SwitchgearBay`) when multiple selections must result in multiple individual instances, each having a quantity of 1.

The main `GeneratorSet` type, inheriting from `LineItem`, typically defaults to `split=none`, where multiple instances are tracked via the quantity attribute on a single line item.

```
// Component type representing a Switchgear Bay (physical slot)
@(split = true)
type SwitchgearBay {
   // Attributes of the split component
   string BayType = ["load", "lv", "mv"];
   int Bay_Number = [1..9];
   int Total_Power_Required_kW = [1..100000];
}
type GeneratorSet {
   // Core attributes used to configure the main product
   @(configurable = false)
   int requiredKW = [101..10000];
   string Voltage = ["220/380", "240/416", "255/440", "277/480",
"347/600", "2400/4160", "7200/12470", "7621/13200",
"7976/13800"];
   decimal(2) surgeLoadKW = requiredKW * 1.25;
   int dBMax = [0..140];
}
```

Purpose of `@split = true`

If multiple instances of the type are created, this setting ensures that the quantity of each instance is always 1.

Example 2: Consolidating Instances (@split = false)

In this example, the `@split = false` annotation is used on component types when you want to ensure that only one instance of that product type exists in the relationship. If the user selects the product multiple times, the constraint engine increments the `quantity` attribute of the existing instance rather than creating a new line item.

```
// Component type where multiple selections consolidate into a
single instance
```

```
@(split = false)
type StandardMaintenanceKit {
   int maintenanceDuration = [12..60];
   string coverageLevel = ["Standard", "Premium"];
}

type GeneratorSet {
   // Core attributes
   @(configurable = false)
   int requiredKW = [101..10000];

   // attribute with an explicit domain
   decimal(2) surgeLoadKW = [126.25..12500.00];

   // Explicit domain
   decimal(2) reserveCapacityKW = [0.00..2500.00];

   string standardsAndCompliance = ["Certification-CSA",
"Listing-UL 2200"];

   // RELATION: split=false ensures all selected kits
consolidate into one instance,updating quantity
   relation MaintenancePlans : StandardMaintenanceKit[0..10];

   // CONSTRAINTS: calculations
   constraint calculateSurgeLoad(surgeLoadKW == requiredKW *
1.25);
   constraint calculateReserve(reserveCapacityKW == surgeLoadKW
- requiredKW);
}
```

Purpose of `@split = false`

This setting ensures that there is only one instance of the type in the relationship. If the user attempts to add more instances, the engine adds more quantity to the existing instance.

Track Components with split Annotation

Consider how you might track components for a large generator order. If a configuration requires three specialized control panels, using `@split = true` (Example 1) gets you three separate line items, each with quantity 1, allowing unique serial numbers or settings for each. Conversely, if you add three identical `StandardMaintenanceKits`, using `@split = false`

(Example 2) ensures that you get one line item for the kit, with a single quantity of 3, keeping your order streamlined for non-unique items.

Example 3: Creating a Virtual Container (@virtual = true)

In this example, the `@virtual = true` annotation is applied to a logical container type, `System`, which is primarily used to define relationships. These relationships aggregate data across line items in the quote that forms a sub-group called `system`. See Relationships.

```
@(virtual = true)
type System {
   // This relation gathers all GeneratorSet line items on the
sales transaction
   @(sourceContextNode =
"SalesTransaction.SalesTransactionItem")
   relation generators : GeneratorSet[0..10];

   // This variable aggregates the surge load (calculated inside
GeneratorSet) from all collected generators
   int totalQuotedLoad = generators.sum(surgeLoadKW);
}

type GeneratorSet {
   // The attribute calculated here is aggregated in the virtual
'System' type above
   @(configurable = false)
   int requiredKW = [101..10000];
   string DutyRating = ["Prime Power (PRP)", "Continuous Power
(COP)", "Data Center Continuous (DCC)", "Emergency Standby Power
(ESP)"];
   decimal(2) surgeLoadKW = requiredKW * 1.25;
}
```

Relationships

Relationships in CML define how different product types are associated with each other, forming the structural hierarchy of a product bundle. Relationships are also referred to as ports.

Here is a comprehensive overview of relationships, their syntax, purpose, and key features, particularly utilizing examples relevant to the Generator Set model.

Definition and Syntax of Relationships

Relationships define the one-to-many connections between a parent type (such as a bundle) and its component types (children).

- Keyword: The keyword used is `relation`.
- Syntax: A basic relationship declaration includes the relation name, the target type, and cardinality bounds.
  `relation <relation name> : <Target Type>[min..max] { /* Optional content */ }`
- Purpose: Relationships represent the product structure in a bundle. For example, the root product (`GeneratorSet`) has relationships with its components (`MainAlternators`, `TemperatureSensors`).

Note: Specifying the smallest required cardinality (quantity range) is a best practice to avoid unnecessary testing of value combinations, which improves performance.

Omit Unnecessary Relationships

When using the visual builder or the CML editor to create a CML code for a bundle, the system by default imports all the relationships for the selected bundle from the structure defined in Product Catalog Management (PCM). In large and complex CML code, some of these relationships may not be relevant to any constraint and can be potentially omitted.

To enable import of a subset of bundle components, add this property at the top of the constraint model CML file:

```
property allowMissingRelations = "true";
```

If your PCM bundle contains many different relations but your CML code defines only one, the engine will validate the model but this often results in a configuration run-time failure. By setting `allowMissingRelations = "true"`, you do not have to define every relation found in the PCM (such as GeneralModels in this example) if they do not require specific configuration logic in your CML file.

allowMissingRelations Example

```
// 1. Enable skipping of unneeded relations from the Product Catalog (PCM)
property allowMissingRelations = "true";

type ConciseGeneratorBundle  {
    // Define only the specific accessory needed for this logic
```

```
    relation enclosures : Enclosure;

    // A simple variable to trigger the logic
    int requiredKW = [100..5000];

    // Logic: High power requirements force a specific enclosure type
    // This omits other accessories like filters, batteries, and
heaters [2, 3].
    constraint(requiredKW > 2000 -> enclosures[ReinforcedEnclosure] ==
1,
               "Power levels above 2000kW require a Reinforced
Enclosure.");
}

// 2. Define the accessory and its specific subtype
type Enclosure ;
type ReinforcedEnclosure : Enclosure;
```

For more information, see Constraints.

To ensure run-time stability without the allowMissingRelations property, you must manually define every single relation and type present in the PCM bundle, even if you don't intend to write logic for them. This creates large CML files with a high number of variables and components, which lead to performance degradation, and even timeout issues.

Note: This code isn't recommended.

```
// EXPENSIVE FIX: Mirroring everything
type GeneratorSet {
    relation engineModels : EngineModel; // Unused in CML logic
    relation alternators : Alternator;    // Unused in CML logic
    relation fuelFilters : FuelFilter;    // Unused in CML logic
    relation starterMotors : StarterMotor; // Unused in CML logic
    relation enclosures : Enclosure;      // The only one we need
    // ... potentially 15+ more relations ...

    constraint(requiredKW > 2000 -> enclosures[ReinforcedEnclosure] ==
1);
}
type Enclosure ;
type ReinforcedEnclosure : Enclosure;
```

Order Keyword

The `order()` keyword is used within a `relation` declaration to define the specific sequence in which the constraint engine evaluates and attempts to instantiate the child subtypes available in that relationship. This controls the prioritization of component selection.

Example: Relationship Ordering

```
// --- Component Subtypes (Specific Generator Models) ---
// Define a base type for generator models with a power
attribute
type GeneralModel {
    int powerKW = [0..3000]; // Explicit domain
}

// Specific subtypes that inherit from GeneralModel
type GeneralModel2500 : GeneralModel {
    int powerKW = 2500;
}

type GeneralModel1750 : GeneralModel {
    int powerKW = 1750;
}

type GeneralModel900 : GeneralModel {
    int powerKW = 900;
}


// --- Parent Type (GeneratorSet) ---
type GeneratorSet {
    // Required power defined by the parent (non-configurable)
    @(configurable = false)
    int requiredKW = [100..3000];

    // Relation Declaration using order()
    // Cardinality requires exactly one model to be selected.
    // order() sets the selection priority (2500 KW model is
checked before 1750 KW model).
    relation GeneralModels : GeneralModel
```

```
        order(GeneralModel2500, GeneralModel1750,
GeneralModel900);
}
```

Relationship Annotations

You can annotate relationships, as in this example, with `configurable=true`.

```
// 1. Define the target type
type Component;
// 2. Define the parent type to hold the relation
type System {
   // 3. Add the relation with cardinality and proper nesting
   @(configurable = true)
   relation components : Component[0..10];
}
```

Here are the details of relationship annotations.

Annotation	Values	Description
allowNewInstance	true, false	Enables require rule constraints to work on relationships that are otherwise closed. Must be set to true to enable require rule constraints on closed relationships.
closeRelation	true, false	If the value is true, prevents the addition of new line items to the relationship. false is the default value.
compNumberVar	true, false	Avoids creating a component number variable if it is set to false.
configurable	true, false	Indicates whether a relationship is configurable. If the value is false, configuration engine doesn’t assign a value to the variable or instantiate a product in the relationship. `true` is the default value.
disableCardinalityConstraint	true, false	Disable cardinality constraint in the relationship to optimize the performance.

domainComputation	true, false	Tells the engine if it needs to compute the domain for the relationship or not.
generic	true, false	Indicates if generic instance is allowed in the relationship or not. Generic instance is used to prompt the user that they need to select a product in the relationship.
noneLeafCardVar	true, false	Avoids creating cardinality variables for none leaf type (a node with no children) in the relationship to optimize the performance.
propagateUp	true, false	Aggregates values from child elements to parent elements.
readOnly	true, false	Sets a relationship and all child relationships to read-only, to prevent the engine or user from modifying.
relatedAttributes	string value	Related attributes for which domain must be reset to original domain for domain computation.
relatedRelationships	string value	Related relationships whose cardinality variables must be unbound for domain computation.
sequence	integer	Indicates the sequence in which the relationship is configured and executed.
sharing	true, false	Indicates if the relationship is shared or not. If the relationship is shared, the engine connects the instance from another relationship to this relationship instead of instantiating the instance in the relationship itself.
singleton	true, false	Indicates if all types in the relationship must be singleton or none.
source	string	Data source defined in the model.
sourceAttribute	Variable name in string	Sets the domain of the current relationship to the domain of the source attribute.

sourceContextNode	string	For cases that use a virtual container, specifies the path in the context service for the instances in the relationship

Constraints

Constraints enforce rules and conditions on types, variables, and relationships. Use constraints to define logical restrictions and ensure consistency within the model. For more information about the supported constraints, see:

- Supported Logic Operators
- Constraint Annotations
- Logical Constraints
- Table Constraint
- Using Proxy Variables with Constraints on Types and Relationships
- Group Type
- Message Rule
- Preference Rule
- Require Rule
- Exclude Rule
- Action Rule
- Hide/Disable Rule

Supported Logic Operators

These logic operators are supported in CML.

Arithmetic Operators

- Multiplication (*)
- Division (/)
- Remainder (%)
- Addition (+)
- Subtraction (-)

Relational Operators

- Greater than (>)
- Greater than or equal to (>=)
- Less than (<)
- Less than or equal to (<=)

Equality Operators

- Equal (==)
- Not equal (!=)

Logic Operators

- Not (!)
- And (&&)
- XOR/Exclusive or (^)
- Or (||)
- Bi-conditional (<->)
- Conditional (?:)
- Implication (->)

Operator Precedence

In resolving equations, operator precedence determines the order in which operations are performed. Operators in CML have precedence in this order:

- Arithmetic operators have the first precedence.
- Relational operators have the second precedence.
- Equality operators have the third precedence.
- Logic operators have a lower precedence than equality operators, in decreasing order as listed, with Implication having the lowest precedence.

Constraint Annotations

Here are the details of constraint annotations.

Annotation	Possible Values	Description
abort	true, false	Specifies that, if this constraint fails, abort search and return false for configuration.
active	true, false	Specifies whether the constraint is active or not.
enddate	date in ISO format	Specifies end date for this constraint. The date is based on the local time in the machine where the configuration service is executed. Date is checked when the configuration is launched. See the Constraints using Format Specifiers (%s, %d) and Dates example.

startdate	date in ISO format	Specifies start date for this constraint. The date is based on the local time in the machine where the configuration service is executed. Date is checked when the configuration is launched. See the Constraints using Format Specifiers (%s, %d) and Dates example.

Logical Constraints

A logical constraint defines a statement that must hold true logically. The constraint can be any logical expression using a logical operator, such as one of these.

For example, the statement `c0 ? c1 : c2` means that if `c0` is true, then `c1` needs to be true, otherwise `c2` needs to be true.

Constraint Syntax Patterns

Here are the details of the constraint syntax patterns.

1. `constraint(logic expression);`: The simplest form, enforcing a logic statement.
2. `constraint(logic expression, string literal | string variable);`: Includes an optional failure explanation that is displayed if the constraint is violated.
3. `constraint(logic expression, string literal | string variable, arg, …, arg);`: Includes a failure explanation with additional arguments to be interpolated into the string of the failure explanation.

If a constraint is violated, it takes an optional string variable or string literal as the failure explanation. The failure explanation could be in a string format with additional arguments. CML supports two string formats. One is the standard string format and another is a string with {} placeholder, to be replaced with the actual argument value.

Key Components of the Constraint Syntax

Here are the details of the key components of the constraint syntax.

Component	Description	Code Sample
Logic Expression	This can be a basic mathematical expression (using operators like `+`, `-`, `*`, `/`) or a relational expression (using	Basic `constraint(generator.sum(quantity) >`

	`==`, `!=`, `>`, `<`, and so on). It can also include logical operators such as AND (`&&`), OR (`||`), NOT (`!`), the implication operator (`->`), or the bi-directional operator (`<->`).	`20);`
Failure Explanation	This optional string literal or variable provides a human-readable reason for the violation. CML supports two formatting styles for these strings: • Java string format (for example, using `%d` or `%s`). • Placeholder format using `{}`.	With Explanation `constraint(x + y <= 100, "Total must not exceed 100");`
Arguments (arg)	Arguments are used to replace the placeholders in the failure explanation string with actual values at runtime.	With Explanation and Arguments `constraint(requiredKW <= 2500, "The required capacity of {} kW exceeds the maximum supported limit of 2500 kW.",requiredKW);`

See the complete example in Constraints using Format Specifiers (%s, %d) and Dates.

Constraint Example

In this example, the first constraint specifies that, if the `DutyRating` value isn't `Prime Power(PRP)`, then the quantity of `Warranty_PRP (Warranties subtype)` must be 0. Similarly in the second constraint, if `DutyRating` value isn't `Data Center Continuous (DCC)then the quantity of Warranty_DCC must be 0`. Lastly, in the third constraint, if `DutyRating` value isn't `Emergency Standby Power (ESP)`, then the quantity of `Warranty_ESP` must be 0.

```
type Warranty;
type Warranty_PRP : Warranty;
type Warranty_DCC : Warranty;
type Warranty_ESP : Warranty;
type GeneratorSet  {
    string DutyRating = ["Prime Power (PRP)", "Continuous Power (COP)", "Data Center Continuous (DCC)", "Emergency Standby Power (ESP)"];
```

```
    relation Warranties : Warranty[3];
    constraint(DutyRating != "Prime Power (PRP)" ->
Warranties[Warranty_PRP] == 0);
    constraint(DutyRating != "Data Center Continuous (DCC)" ->
Warranties[Warranty_DCC] == 0);
    constraint(DutyRating != "Emergency Standby Power (ESP)" ->
Warranties[Warranty_ESP] == 0);
}
```

Example: Constraints Using String and Logical Operators

This example demonstrates how a `GeneratorSet` uses string functions to extract a numeric voltage value, and then uses logical operators (->, &&, ||) to enforce safety and certification requirements.

Explanation of Operators Used in the Example

Here are the details of the operators used in the example.

Operator/Function	Category	Usage in Example
`strtoint()`	String Function	Converts the extracted voltage string to the integer `Voltage3`.
`regexpreplace()`	String Function	Replaces the full `Voltage` string with only the second matched group (the high voltage number) using the defined `VOLTAGE_REGEX`.
`strcontain()`	String Function	Checks if the `DutyRating` string contains the substring "Power".
`->`	Logical Implication	Defines a conditional rule: If the condition on the left is true, the condition on the right must be true.
`&&`	Logical AND	Requires both conditions (high requiredKW AND duty rating contains "Power") to be true.
`||`	Logical OR	Requires `standardsAndCompliance` to be `"Listing-UL 2200"` or `"Certification-CSA"`.

`!=`	Comparison/ Relational	Used to check if the string value is "Not Equal" to a specific string literal.

```
// --- Constants (Required for String Manipulation) ---
define VOLTAGE_REGEX "^(+)/(+)$"

// --- Base Type and Configuration Attributes ---
type Warranty;
type Warranty_ESP : Warranty;

type GeneratorSet {
    // String input attribute for user selection
    string Voltage = ["277/480", "2400/4160", "7976/13800"];

    // String input attribute for operational mode
    string DutyRating = ["Prime Power (PRP)", "Continuous Power
(COP)", "Emergency Standby Power (ESP)"];

    // String input attribute for compliance
    string standardsAndCompliance = ["Certification-CSA",
"Listing-UL 2200"];
    // Required KW (Int attribute)
    int requiredKW = [101..10000];
     // warranties for generator set
     relation Warranties : Warranty[3];

    // 1. STRING OPERATORS/FUNCTIONS: Deriving Numeric Data
    // We use strtoint() combined with regexpreplace() to
extract the second number (the high voltage)
    // from the Voltage string (e.g., extracting 480 from
"277/480").
    int Voltage3 = strtoint(regexpreplace(Voltage,
VOLTAGE_REGEX, "$2"), 0);
  // Prime Power or Continuous Power using strcontain
    boolean NonstandbyPower = !strcontain(DutyRating, "ESP");

    // 2. LOGICAL OPERATOR (Implication ->): Certification
Validation
    // Constraint: If the standard is UL 2200 (precondition),
THEN Voltage3 must be <= 600 (postcondition).
```

```
    constraint(
        standardsAndCompliance == "Listing-UL 2200" -> Voltage3
<= 600,
        "The UL 2200 standard covers stationary engine generator
assemblies rated at 600 volts or less."
    );


    // 3. LOGICAL OPERATORS (AND &&, OR ||) and STRING FUNCTION
(strcontain)
    // Constraint: If the required power is high AND the unit is
used for Prime Power OR Continuous Power,
    // THEN a specific standard must be selected.
    constraint(
        (requiredKW >= 5000 && NonstandbyPower)
        ->
        (standardsAndCompliance == "Listing-UL 2200" ||
standardsAndCompliance == "Certification-CSA"),  "High power and
continuous use requires a major compliance standard.");


    // 4. LOGICAL OPERATOR (Implication ->) and String
Comparison (!=)
    // Constraint: If the Duty Rating is NOT Emergency Standby
Power, THEN a specific warranty (Warranties[Warranty_ESP]) must
be excluded/zero.
    constraint(
        DutyRating != "Emergency Standby Power (ESP)" ->
Warranties[Warranty_ESP] == 0,
        "The DutyRating when not equal to Emergency Standby
Power, implies that the Warranty must be 0."
    );
}
```

How User Input Order Affects Constraint Engine Behavior

The constraint engine is architecturally designed to never override or modify a user-selected value when evaluating constraints, except in the specific case of an `exclude` rule. If a constraint violation occurs due to user input, the engine generates an error message rather than attempting to fix the value itself.

Example: Constraint Evaluation Based on Input Order (Generator Set)

The order in which a user configures attributes for a product (like a `GeneratorSet`) determines whether the constraint engine performs an automatic update or enforces a validation error.

Consider a constraint within the `GeneratorSet` type that links the operational requirement (`DutyRating`) to the required certification (`standardsAndCompliance`).

```
type GeneratorSet {
    // Attribute 1 (Precondition): User selects operational mode
    string DutyRating = ["Prime Power (PRP)", "Continuous Power
(COP)"];
    // Attribute 2 (Dependent): Certification standard
    string standardsAndCompliance = ["Certification-CSA",
"Listing-UL 2200"];
    // Constraint: If the unit is configured for Prime Power, it
must have UL 2200 Listing.
    constraint(
        DutyRating == "Prime Power (PRP)" ->
standardsAndCompliance == "Listing-UL 2200",
        "Prime Power Duty Rating requires UL 2200 Listing"
    );
}
```

The outcomes depend on the user's input sequence.

Scenario	User Input Sequence	Constraint Engine Result
Scenario 1 Successful Update (Precondition first)	The user first sets `DutyRating` to `"Prime Power (PRP)"`.	The constraint engine recognizes the precondition is met and updates the dependent attribute, setting `standardsAndCompliance` to `"Listing-UL 2200"`. The constraint is validated.
Scenario 2 Constraint Violation (Conflicting Value first)	The user first sets `standardsAndCompliance` to `"Certification-CSA"`,	The existing user-selected value (`"Certification-CSA"`) violates the constraint. The

	and then sets `DutyRating` to `"Prime Power (PRP)"`.	constraint engine will not override the user's prior selection to change it to `"Listing-UL 2200"`. Instead, the engine displays an error.

To resolve the error in Scenario 2, the user must manually adjust one of their selections: either change the `DutyRating` (precondition) or manually update the `standardsAndCompliance` (dependent value) to `"Listing-UL 2200"`.

Left-Hand Side and Right-Hand Side Behavior in Constraint Resolution

Operator precedence and constraint engine resolution process determines whether the left-hand side (LHS) or right-hand side (RHS) of a constraint changes or is constrained to maintain logical validity. Variable origin, which means whether variables are user-selected, calculated, or restricted by system limitations, can also affect the outcome for the LHS or RHS in an expression.

These examples show how different user inputs lead to different outcomes for the LHS and RHS in the same expression.

Example 1. Implication Operator (->): Directional Enforcement

```
type Order {
    int quantity = [1..1000];
    boolean requiresApproval;
    constraint bulkOrderApproval (
        quantity >= 100 -> requiresApproval == true
    );
}
```

The implication constraint `A -> B` (if A, then B) is the primary method for defining a mandatory outcome. The engine evaluates the LHS (A) first. If the `quantity` is 150 (LHS TRUE), then the engine forces `requiresApproval` (RHS) to be TRUE.

Scenario	Outcome
Scenario A LHS is True, RHS Changes	The user sets a quantity greater than or equal to 100 on the LHS, which makes the condition true. The engine sets or changes the value of requiresApproval, the RHS, to true.
Scenario B RHS is False, LHS is	The user manually sets requiresApproval, the RHS variable, to false. Since the implication true -> false is forbidden, the LHS

Constrained to be False	condition must be false to satisfy the constraint. The engine constrains quantity on the LHS to be less than 100 to make the LHS false.

Example 2. Bi-conditional (<->): Symmetrical Equivalence

```
type BulkOrderSystem {
    int quantity = [1..1000];
    boolean requiresApproval;
    boolean isBulkOrder;

    constraint bulkOrderStatus (
        isBulkOrder <-> (quantity >= 100 && requiresApproval),
        "Bulk order status requires 100+ quantity AND management
approval."
    );
}
```

The bi-conditional constraint `A <-> B` (A if and only if B) requires that the LHS and RHS must share the same Boolean truth value. Either side can act as the driver, forcing the other side to change. In the example above:

- A is the boolean variable `isBulkOrder`.
- B is the complex condition `(quantity >= 100 && requiresApproval)`.

Scenario	Outcome
Scenario A LHS Drives RHS	If the user manually sets the attribute isBulkOrder to true (making the LHS true), the engine immediately forces the RHS (quantity >= 100 && requiresApproval) to also be true, ensuring that both quantity is at least 100 and requiresApproval is set to true.
Scenario B RHS Drives LHS	If the user selects a configuration where the quantity is high (for example, 500) and then sets requiresApproval to false (making the RHS condition false), the engine immediately forces the LHS attribute isBulkOrder to false to maintain equivalence.

Exception: The exclude Rule

The only scenario where the constraint engine intentionally overrides user input is when processing the `exclude` rule. If a user selects a component or sets an attribute value that violates an `exclude` rule, the engine will automatically override that user input to satisfy the

exclusion constraint. In all other constraint types (like implication constraints shown previously), the engine relies on the user to fix the error. For more information, see Exclude Rule.

Table Constraints

The `table` constraint in CML is used to define a set of valid combinations of values for two or more attributes. These combinations are specified in rows within the constraint definition.

The table constraint has this syntax:
`table(variable, …, variable, {value, .. value}, …, {value, …, value});`

Each row inside `{}` defines a valid combination of values.

Example: Table Constraint

In this example:

- Variables: The attributes `Voltage` and `DutyRating` are listed as the columns for the table.
- Table Rows: Each row defined within the curly braces (`{}`) specifies a valid combination. For instance, `{"7976/13800", "Continuous Power (COP)"}` is a valid pairing.
- Enforcement: If a user attempts to select a high voltage (`"7976/13800"`) while choosing a Prime Power rating (`"Prime Power (PRP)"`), the table constraint is violated, and the engine displays the error message: "Selected Voltage is not compatible with the required Duty Rating."

```
// --- Component Types ---

type GeneratorSet {
    // 1. Attributes whose values must align according to the table
    string Voltage = ["220/380", "277/480", "7976/13800"];
    string DutyRating = ["Prime Power (PRP)", "Continuous Power (COP)", "Emergency Standby Power (ESP)"];
    // 2. Table Constraint
    // Defines valid combinations where Voltage and DutyRating are mutually dependent.
// Combination 1: Low Voltage is compatible with all operational modes
```

```
 // Combination 2: Standard US Voltage (277/480) requires Prime
or Emergency duty.
// Combination 3: High Voltage (7976/13800) is only compatible
with Continuous duty.
    constraint validOperationalModes(
        table(
            Voltage,
            DutyRating,
                                {"220/380", "Prime Power
(PRP)"},
                {"220/380", "Continuous Power (COP)"},
                {"220/380", "Emergency Standby Power (ESP)"},
                {"277/480", "Prime Power (PRP)"},
                {"277/480", "Emergency Standby Power (ESP)"},
                {"7976/13800", "Continuous Power (COP)"}
        ),
        "Selected Voltage is not compatible with the required
Duty Rating."
    );
}
```

Import Data from a Salesforce Object to Populate a Table Constraint

Import data from a standard or custom Salesforce object to use in a table constraint in a constraint model. The imported data populates the columns and rows in the table constraint in CML, and saves you the step of manually entering the data.

To import data from a Salesforce object, first assign Read, Create, Edit, and Delete permissions for the object to the Constraint Rules Engine Licenseless permission set. See Import Data from Salesforce Objects to Use in Constraint Models in Salesforce Help.

In CML, use the SalesforceTable keyword and the syntax shown here to import data from a Salesforce object. This example uses the `GeneratorSet` type to constrain the calculated running capacity (`gc_runningKw`) based on a user's selection of the nominal output (`Nominal_Power_Output`), referencing an external Salesforce custom object named PowerCst__c.

Example: Imported Table Constraint

```
type GeneratorSet {
    // 1. Attribute storing the user-selected power output
(String)
```

```
    string Nominal_Power_Output = ["100 kW", "300 kW", "500 kW",
"700 kW"];

    // 2. Attribute storing the resulting Running kW (Decimal,
calculated)
    @(configurable = false, defaultValue = "0")
    decimal(2) gc_runningKw;
    // Constraint ensures the pairing of Nominal_Power_Output
and gc_runningKwis found in the imported Salesforce table.
    constraint(
        table(
        Nominal_Power_Output,
          gc_runningKw,
          SalesforceTable( "PowerCst__c","Nominal__c,Running__c"
)
)
);
}
```

Explanation of the Imported Table

The table constraint ensures that the selected values for `Nominal_Power_Output` and `gc_runningKw` must form one of the valid combinations defined in the external source.

1. Table (`Nominal_Power_Output`, `gc_runningKw`, ...): These are the CML attributes whose values must correlate. They define the columns of the required combination table.
2. Salesforce Table ("PowerCst__c", "Nominal__c,Running__c"): This function keyword directs the constraint engine to import data from the Salesforce custom object PowerCst__c.
3. Field Mapping: The fields specified ("`Nominal__c,Running__c`") define the columns in the Salesforce object that correspond to the CML attributes listed in the table function. Nominal__c maps to `Nominal_Power_Output`, and Running__c maps to `gc_runningKw`.

Using Proxy Variables with Constraints on Types and Relationships

Use proxy variables to reference the variables of related types, including parent, root, and sibling types. For more information about the supported proxy variables, see:

- Cardinality
- Parent
- this.quantity

Cardinality

Cardinality is a fundamental concept in CML, controlling the quantity of product instances allowed in a defined relationship. It is crucial for ensuring product structure validity and optimizing performance.

The `cardinality` proxy variable refers to the cardinality of a relationship, that is, the quantity of instances of the same type in a relationship. The first parameter is the type name. The second, optional parameter is the port name. This variable differs from the `this.quantity` proxy variable, which refers to the quantity of the current instance.

Use this format:
`cardinality(<type name>, <relation name>) or cardinality(<type name>)`

Each parameter value can be a string or a string variable. If the relation name isn't specified, the engine searches all ports to find the type.
The cardinality bounds are defined within the `relation` declaration using square brackets, formatted as `[minimum..maximum]`.

Cardinality Examples

These examples illustrate how cardinality is defined and managed in the Generator Set model, focusing on standard definition, conditional enforcement, and reading the quantity.

Example 1: Using cardinality full syntax

```
type Heater ;
type GeneratorSet  {
  //Define the relation and specify the smallest cardinality
required for performance
    relation Heaters : Heater[0..10];
    //Declare a variable to hold the count
    int totalHeaterCount = [0..10];
    // This counts instances of type "Heater" specifically
within the "Heaters" relation.
    int InstancesofHeater = cardinality(Heater, Heaters);
    constraint(totalHeaterCount == InstancesofHeater);
    // Optional message to display the count to the user
    message(totalHeaterCount > 0, "Current configuration
includes {} heaters.", totalHeaterCount, "Info");
}
```

See Message Rule.

Example 2: using cardinality partial syntax

```
type Heater ;
type OutputTerminal ;

// Main Generator Set type
type GeneratorSet {

    // Define multiple relations that might contain specific
types, each with a specified cardinality
    relation Heaters : Heater[2];
    relation OutputTerminals : OutputTerminal[0..99];

    // Define derived attributes with an explicit domain
    int totalHeatersFound = [0..100];
    int totalTerminalsFound = [0..100];

    // Partial cardinality SYNTAX
    // The engine searches all relations to find the specified
type

    // Counts all instances of "Heater" across the entire
GeneratorSet
    int AllHeaterInstances = cardinality(Heater);
    constraint(totalHeatersFound == AllHeaterInstances);

    // Counts all instances of "OutputTerminal" across all
relations
    int AllOutputTerminals = cardinality(OutputTerminal);
    constraint(totalTerminalsFound == AllOutputTerminals);

    // Optional message for user visibility
    message(totalHeatersFound > 0, "System found {} heaters in
this configuration.", totalHeatersFound, "Info");
}
```

Example 3: Defining Cardinality in Relations

This example shows how the Generator Set uses different cardinality ranges to define fixed requirements, optional components, and minimum quantities for necessary components.

```
type GeneratorSet {
    // 1. Fixed Cardinality: Exactly one General Model
(component) is required.
    // means min=1 and max=1.
    relation GeneralModels : GeneralModel;

    // 2. Bounded Optional Cardinality: Between 0 and 5
Temperature Sensors are allowed.
    // [0..5] means the component is optional but limited.
    relation TemperatureSensors : TemperatureSensor[0..5];

    // 3. Minimum Required Cardinality: At least 1 Test record
is required, up to 99.
    // [1..99] enforces inclusion.
    relation Testing : Test[1..99];
}

type GeneralModel;
type TemperatureSensor;
type Test ;
```

Key Concepts

- Syntax: Cardinality is specified immediately after the component type in the relation declaration.
- Best Practice: Specifying the smallest required range, such as `` or `[0..5]`, ensures the constraint engine tests fewer combinations, which prevents performance degradation.

Example 4: Enforcing Conditional Cardinality via require()

Cardinality can be dynamically enforced based on attributes of the parent product using the `require()` constraint keyword. For more information, see Require Rule.

```
type GeneratorSet {
    int requiredKW = [101..10000];
    string standardsAndCompliance = ["Certification-CSA",
"Listing-UL 2200"];

    // Relation for mandatory installation accessories
    relation Accessories : Accessory[1..99];
    relation Testing : Test[1..99];
```

```
    // Conditional Requirement based on power level:
    // If the required power is over 5000 kW, exactly 5 specific
Accessory instances are required.
    require(
        requiredKW > 5000,
        Accessories [Accessory] == 5,
        "High capacity generators require exactly 5 specialized
accessory kits."
    );

    // Conditional Requirement based on compliance standard:
    // If UL 2200 is selected, exactly 2 Test records must be
included.
    require(
        standardsAndCompliance == "Listing-UL 2200",
        Testing [Test] == 2,
        "UL 2200 listing mandates exactly two certification
tests."
    );
}
type Accessory;
type Test;
```

Purpose

This pattern (`require(condition, relation[type] == N, ...)`) allows the CML model to enforce a precise fixed number of child components when a quantity threshold or attribute condition is met on the parent.

Cardinality vs. Count

The `cardinality()` keyword is a proxy variable used to refer to the size of a relationship. The `count()` keyword is an aggregate function that counts matching components or attribute conditions within a relation. Both can be used to read quantities for validation or aggregation rules.

Example for Similarity

Both `cardinality()` and `count()` can be used to determine the total quantity of items in a relationship. In this scenario, they return the same value because the `count()` condition covers all possible active instances.

```
type Accessory{
        boolean isPresent = true;
}

type Bundle {
    relation items : Accessory[0..10];
    int totalByCount = [0..10];
    // Similarity: Both can read the headcount of the relation
    // Use the proxy variable to set an attribute
    int totalByCardinality = cardinality(Accessory, items);
    // Use count() aggregate function within a constraint
    constraint(totalByCount == items.count(isPresent == true));

}
```

Example for Difference

The core difference is that cardinality is a proxy variable representing the headcount/size of the relation, whereas count is a function used to evaluate specific attribute conditions or filters within those instances

```
type Accessory {
    int weight = [1..100];
    boolean isEssential;
}

type Bundle {
    relation items : Accessory[1..20];

    // 1. Declare variables with explicit domains [1, 2]
    int totalHeadcount = cardinality(Accessory,items);
    int heavyItemsCount = [0..20];
    int essentialItemsCount = [0..20];

    // DIFFERENCE 1: Cardinality (Proxy Variable)
    // Cardinality refers to the relationship size (how many instances are present) [3, 4].
    // It does not look at the values of the attributes within the instances.

    // DIFFERENCE 2: Count (Aggregate Function)
```

```
    // Count MUST use a logical expression to evaluate
conditions within the relation [5, 6].
    // It filters the instances based on attribute logic before
returning a total.

    // Example: Counting only accessories that weigh more than
50kg
    constraint(heavyItemsCount == items.count(weight > 50));

    // Example: Counting only accessories marked as 'essential'
    constraint(essentialItemsCount == items.count(isEssential ==
true));

    // Business Logic using both:
    message(
        heavyItemsCount > (totalHeadcount / 2),
        "Warning: More than half of your accessories ({}) are
heavy items.",
        heavyItemsCount
    );
}
```

For more information, see Message Rule.

Parent

The `parent()` proxy variable is used to enable components at any level of the product hierarchy (child, grandchild, etc.) to access the attributes (variables) defined by their ancestor types. This mechanism facilitates the flow of configuration data and calculated values from the top of the bundle down to its components.

The `parent()` keyword functions as a proxy variable used to reference attributes residing in the immediate parent or any ancestor type in the configuration model.

Variable Name	Syntax	Purpose
`parent()`	`parent(<ancestor variable name>, <level>)`	Retrieves the value of an attribute from a parent or ancestor type.

Key Characteristics

- Targeting Ancestors: The first parameter is the name of the attribute in the ancestor type that you wish to reference.
- Level Parameter: The second parameter (`<level>`) is optional and specifies how many levels up the hierarchy the engine should search. If omitted, it typically references the immediate parent. The level index for the `parent()` function effectively starts from 0 (implicit) for the immediate parent.
    - Level 0 (Default): When you use `parent(attributeName)`, the engine references the attribute in the immediate parent.
    - Level n: When you specify `parent(attributeName, 1)`, the engine reaches one level beyond the immediate parent, to the grandparent.
- Data Flow: `parent()` ensures that the data flows unidirectionally, where the child reads attributes calculated or defined by the parent, thereby helping to prevent complex circular dependencies.
- Single Inheritance: CML follows a single inheritance model, where a type can only extend one other type at a time.
- Same Variable Name: In CML, reusing the same variable name between a parent bundle and its child accessories is a standard architectural pattern for cascading values down a product hierarchy. This allows a child component to automatically inherit or synchronize its properties with the parent type, ensuring configuration consistency.

Example: Standards and Compliance Synchronization

In this example, the `standardsAndCompliance` selection made at the bundle level is automatically passed down to every accessory within that bundle.

```
type AccessoryBundle {
    // 1. Define the attribute at the Bundle level
    string standardsAndCompliance = ["Certification-CSA",
"Listing-UL 2200"];
    // Relation to accessories
    relation accessories : Accessory[1..10];
}
type Accessory {
    // 2. Reuse the same variable name
    // The parent() function targets the attribute in the
immediate parent
    string standardsAndCompliance =
parent(standardsAndCompliance);
```

```
    // Business Logic: If the inherited standard is UL 2200,
price is affected
    decimal(2) testingFee = [0.00..500.00];
    constraint(standardsAndCompliance == "Listing-UL 2200" ->
testingFee == 250.00);
}
```

The `parent()` proxy variable is essential for cascading configuration data and constraints down the product hierarchy, allowing child components to reference attributes defined by their ancestors.

Example: Parent Attribute Reference

This example utilizes attributes found on the `GeneratorSet` type (the parent) and demonstrates how related components (like TemperatureSensor) access those values using the parent() proxy variable. Here is the hierarchy context.

Parent Type	Relation Name	Child Type	Cardinality	Key Attribute Usage
Root `GeneratorSet`	`TemperatureSensors`	`TemperatureSensor`	`[0..5]`	Defines primary inputs (`requiredKW`) and a derived attribute (`ULComplianceEnforced`) based on the `standardsAndCompliance` selection.
Child `TemperatureSensor` (Instance within `GeneratorSet` relation)	Not Applicable	Not Applicable	Not Applicable	Uses the `parent()` proxy variable to reference the `requiredKW` attribute from its ancestor (`GeneratorSet`). A constraint ensures the component's capacity (`maxOperatingKW`) meets the requirement inherited from the parent.

Grandchild `StatorTemperatureSensor` (Subtype of `TemperatureSensor`)	Not Applicable	Not Applicable	Not Applicable	Inherits functionality from `TemperatureSensor`. Uses parent() to retrieve the calculated `ULComplianceEnforced` boolean from the `GeneratorSet`. Enforces a conditional installation rule using the Implication Operator (`->`).

```
// --- Parent Type (GeneratorSet) ---
type GeneratorSet {
    // Attributes defined by the parent
    int requiredKW = [101..10000]; // Required power rating
    string standardsAndCompliance = ["Certification-CSA",
"Listing-UL 2200"]; // Compliance choice

    // Derived status: True if UL 2200 is selected (precondition
for compliance checks)
    boolean ULComplianceEnforced = standardsAndCompliance ==
"Listing-UL 2200";

    // Relation to child components
    relation TemperatureSensors : TemperatureSensor[0..5];
}

// --- Child Component Type (TemperatureSensor) ---
type TemperatureSensor {
    // Temperature sensors may have a model that defines max
operating KW
    int maxOperatingKW = [1000..10000];

    // Retrieve the required KW value from the immediate parent
(GeneratorSet)
    int parentRequiredKW = parent(requiredKW);
```

```
    // Constraint ensures the sensor can handle the power
defined by the parent
    constraint(maxOperatingKW >= parentRequiredKW, "Sensor
capacity must meet the required KW set by the generator set.");
}

// --- Specific Sensor Type (Grandchild) ---
type StatorTemperatureSensor : TemperatureSensor {
    // Retrieve the boolean compliance flag from the immediate
parent (GeneratorSet)
    boolean enforceULCompliance = parent(ULComplianceEnforced);

    // Constraint ensures that if UL Compliance is enforced by
the parent,
    // this sensor must meet a specific installation requirement
(e.g., must be shielded)
    string installationMethod = ["Standard", "Shielded"];
    constraint(enforceULCompliance -> installationMethod ==
"Shielded",
        "UL Compliance requires a shielded temperature sensor
installation."
    );
}

// Note: To reference a value further up the hierarchy, the
optional level parameter can be used:
// int grandParentValue = parent(requiredKW, 1); // Accesses
attribute 1 level up
```

Explanation of Parent Reference

- Direct Reference (Immediate Parent): In the `TemperatureSensor` type, `int parentRequiredKW = parent(requiredKW);` accesses the `requiredKW` attribute defined in the immediately superior type, which is `GeneratorSet`.
- Unidirectional Data Flow: The `parent()` proxy variable is critical for enabling unidirectional data flow, where calculated or defined values in the parent are read by children to enforce constraints. The engine ensures that parent aggregation and calculation are complete before the `parent()` function executes in the child component.

this.quantity

`this.quantity` is a proxy variable used specifically to access the quantity of the current instance (the specific product line item) within a configuration.

Key Characteristics and Usage

- Scope: It refers only to the quantity of the specific instance in which it is used. It is used at the component level to determine the quantity of that item chosen by the user or set by the constraint engine.
- Calculation Rule**:** The `this.quantity` proxy variable can be used only within a calculation rule.
- Distinction from `cardinality()`: `this.quantity` differs from the `cardinality()` proxy variable, which refers to the total number of instances of a specific type within a relationship and includes the quantity of other instances of the same type.
- Read-Only/Validation: When accessing quantity in CML, attributes like `this.quantity`, or external equivalents like `lineItemQuantity` or `ItemEndQuantity`, are treated as read-only and should be used only in calculation or evaluation rules. It is not recommended to use it to drive component creation, which should be done via cardinality.

Example: Using this.quantity for Calculation

In the Generator Set configuration, a component like the `NaturalGasReformer` may use `this.quantity` to define a local attribute representing how many units were selected, which is then used for internal calculations (like total capacity or total weight contribution).

```
type LineItem;
type NaturalGasReformer : LineItem {
    // 1. Definition: The LineItemQuantity attribute captures
the quantity of this specific instance.
    // The CML syntax dictates defining an attribute
(LineItemQuantity) that equals this.quantity.
    int LineItemQuantity = this.quantity;

    // Placeholder: Attribute representing unit capacity per
reformer unit
    int unitCapacityKW = 100;
```

```
    // 2. Calculation: Used in a calculation rule to determine
total capacity based on the quantity selected for this line item
instance.
    int totalReformerCapacity = LineItemQuantity *
unitCapacityKW;
}
```

Group Type

In CML, a Group Type is used to logically containerize related components within a bundle configuration. This structure is primarily utilized when product component groups are imported from Product Catalog Management (PCM). The Group Type uses specific annotations to control the total quantity of components selected from that group, regardless of the individual component type.

Bundles and Group Types (also known as Product Component Groups or PCGs) represent different levels of a product hierarchy and serve distinct functional roles in configuration logic.

Conceptual Hierarchy

- Bundles are high-level parent products that contain multiple child products sold together as a package. In CML, they are defined as "root types" that encapsulate the properties, relationships, and constraints for the entire entity.
- Group Types are structural containers within a bundle. They act as intermediate folders that organize related components imported from Product Catalog Management (PCM). Instances of these Group Types are declared as variables inside the root Bundle type.

Role in Cardinality and Selection

- Bundles establish the primary relationship between a root product and its broad categories of components.
- Group Types are specifically designed to enforce cardinality rules for a collection of products. They use the `@(minInstanceQty)` and `@(maxInstanceQty)` annotations to control exactly how many instances can be selected from a specific set of options (for example, "select at least 1 but no more than 2 accessories"). While the selected component can have a high quantity, the Group Type restricts the number of unique instances chosen from that group.

Syntactic Implementation

- Accessing Components: For standard bundles, you reference components directly via their relation name. For components within a Group Type, you must use dot notation

starting with the group variable name defined in the root type (for example, `accessoryGroup.mouse.Wireless == true`).

- Constraint Limitations: You cannot write a constraint directly on a Group Type's attribute to apply it to all components within that group; constraints must reference the specific child components.
- Identification: Group Types are automatically identified by a Group suffix and the presence of instance cardinality annotations during the import from PCM.

Note: The following examples are partial. See the complete code in the final CML code sample.

Example 1: Defining Generator Set Group Types

We can define two group types for a Generator Set configuration: `CoreModelGroup` (mandatory selection of a primary generator model) and `EnclosureGroup` (optional selection of a specific enclosure type).

For the `CoreModelGroup`, setting `minInstanceQty` to 1 and `maxInstanceQty` to 1 means that at least 1 is required, and a maximum of 1 is permitted.

```
@(minInstanceQty=1, maxInstanceQty=1)
type CoreModelGroup {
    // Relation holds the actual Generator Model products
    relation generalModel : GeneralModel[0..9999];
}

type GeneralModel : LineItem {
    int powerKW;
}
```

For the `EnclosureGroup`, setting `minInstanceQty` to 0 and `maxInstanceQty` to 1 means that selecting an enclosure is optional, but if selected, the user can add at most one instance of an enclosure component (such as WeatherproofEnclosure).

```
@(minInstanceQty=0, maxInstanceQty=1)
type EnclosureGroup {
    relation enclosure : Enclosure;
    relation cabinetHeater : ControlCabinetHeater;
}

type Enclosure : LineItem;
type ControlCabinetHeater : LineItem;
```

Example 2: Referencing Group Types in the Root Bundle

Once defined, instances of these group types are used as variables within the root type, which represents the entire bundle. In this example, the `coreModelGroup` and `enclosureGroup` are instances of the respective group types, defined within the root type `GeneratorSetBundle`.

```
type GeneratorSetBundle {
    CoreModelGroup coreModelGroup;
    EnclosureGroup enclosureGroup;
}
```

Example 3: Writing Constraints on Group Components

To write constraints or rules that reference components inside a group, use dot notation starting with the group variable name defined in the root type.

This example shows how a constraint might be defined within the `GeneratorSetBundle` type to enforce business logic on components within the groups.

```
// If the selected generator model has a powerKW greater than
1500,
// then a Control Cabinet Heater must be included in the
Enclosure Group.
constraint(coreModelGroup.generalModel.powerKW > 1500 ->
           enclosureGroup.cabinetHeater[ControlCabinetHeater] ==
1);

// Require an Enclosure component if the set requires seismic
certification.
require(seismicCertification == "IBC Seismic Certification",
        enclosureGroup.enclosure[Enclosure],
        "Enclosure required for seismic certification");
```

Final CML Code Sample with Group Types

This structure defines the complete model, showing the component hierarchy and the relationship constraints.

Parent Type	Relation Name	Child Type	Cardinality	Key Attribute Usage
`GeneratorSetBundle`	`coreModelGroup` (Group Instance)	`CoreModelGroup`	Group Cardinality: 1	The group type defined by `@minInstanceQty=1, maxInstanceQty=1`. This enforces that exactly one component choice must be made from the internal generalModel relation.
`GeneratorSetBundle`	`enclosureGroup` (Group Instance)	`EnclosureGroup`	Group Cardinality: [0..1]	The group type defined by `@minInstanceQty=0, maxInstanceQty=1`. This enforces that selecting components from this group is optional (min 0) and at most one total component instance can be selected (max 1).

```
/**
 * The Root Bundle: GeneratorSetBundle
 * This type holds instances of the defined Group Types and the
definition of seismic certification */
type GeneratorSetBundle {
    string seismicCertification = ["IBC Seismic Certification",
"OSHPD Seismic Certification"];

    CoreModelGroup coreModelGroup;
    EnclosureGroup enclosureGroup;

    // Constraint 1: If the selected GeneralModel has a powerKW
greater than 1500,
    // then a Control Cabinet Heater must be included in the
Enclosure Group.
    constraint(coreModelGroup.generalModel.powerKW > 1500 ->

enclosureGroup.cabinetHeater[ControlCabinetHeater] == 1);
```

```
    // Constraint 2: Require an Enclosure component if the set
requires seismic certification.
    require(seismicCertification == "IBC Seismic Certification",
            enclosureGroup.enclosure[Enclosure],
            "Enclosure required for seismic certification");

    // Action Rule Example: Disable a specific enclosure type if
the core model is low power (e.g., 900kW).
    rule(coreModelGroup.generalModel.powerKW == 900, "disable",
"relation",
         "enclosureGroup.enclosure", "type",
"ReinforcedEnclosure");
}

/**
 * Group Type 1: Core Model Group (Mandatory, Single Select)
 * Min/Max Instance Quantity controls the selection rules for
components within this group.
 */
@(minInstanceQty=1, maxInstanceQty=1)
type CoreModelGroup {
    // Relation holds the actual Generator Model products
    relation generalModel : GeneralModel[0..9999];
}

/**
 * Group Type 2: Enclosure and Accessories Group (Optional)
 * minInstanceQty=0 means selection is optional.
maxInstanceQty=1 limits the selection to a single enclosure or
accessory component instance.
 */
@(minInstanceQty=0, maxInstanceQty=1)
type EnclosureGroup {
    relation enclosure : Enclosure;
    relation cabinetHeater : ControlCabinetHeater;
}

/**
 * Component Types (Children)
 */
type GeneralModel {
```

```
    int powerKW;
}
type Enclosure {
    @(defaultValue = "false")
    boolean Weatherproof;
}
type ReinforcedEnclosure : Enclosure; // Subtype of Enclosure
type ControlCabinetHeater;
```

Key Considerations

When reviewing Group Types in the context of the Generator Set model, keep these architectural points in mind.

- Group Cardinality Enforcement: The annotations `@(minInstanceQty)` (minimum instance quantity) and `@(maxInstanceQty)` (maximum instance quantity) are defined on the Group Type itself (`ElectricalSafetyGroup`). These annotations control the overall cardinality for all the components contained within that group, regardless of the individual relation cardinality defined (for example `[0..1]`, `[1..2]`).
- Root Reference: The `GeneratorSetBundle` includes the groups as variables (`coreModelGroup` and `enclosureGroup`).
- Constraint Syntax for Group Components: Constraints access attributes or components inside the groups using dot notation starting with the group variable name (for example, `coreModelGroup.generalModel.powerKW`).
- Limitation on Group Attributes: You cannot write a constraint directly on a group's attribute and expect it to apply to all components within that group (for example, `constraint(enclosureGroup.color == "Black")`) is not a valid constraint).

Message Rule

The message rules display a message to users based on specified conditions.

Message rules have this syntax:

```
message(logical expression, string literal | string variable,
argument, .., argument, severity);
message(logical expression, string literal | string variable,
severity);
message(logical expression, string literal | string variable);
```

A message rule can take optional arguments to generate the message and indicate the severity of the message as the last argument. Message severity can be `Info`, `Warning`, or `Error`. Without an explicit message severity argument, the message will be treated as `Info`.
Understand the behaviour of each message severity type at runtime.

- The `Info` message type doesn't require the user to take any action in order to continue with the current task. `Info` messages display a gray banner.
- The `Warning` message type allows the user to continue working on the current task, but blocks them from taking the next step until they take action to address the issue described in the message. `Warning` messages display a yellow banner.
- The `Error` message type blocks the user from continuing with the current task until they fix the error described in the message. `Error` messages display a red banner.
  Note: An `Error` message doesn't block a user working in the Transaction Line Editor (Transaction Line Table, or TLT). In that component, the user can still make changes and save the quote, even when the quote contains conditions that trigger an `Error` message.

Message format can be a Java string, or a string with {} as a placeholder. The constraint solver replaces each {} with arguments specified after the string, in the order they are written.

In this example, if `requiredKW` is greater than `2500`, a message is displayed that the specified required kW is larger than the supported options and must be changed.

```
type GeneratorSet {
   int requiredKW = [101..10000];
message(requiredKW > 2500, "The required kW is above what the
current options can support. Please adjust to 2500 kW or select
a new generator set that meets your requirements.");
}
```

Preference Rule

The preference rule encourages the constraint solver to satisfy the condition, but doesn't enforce it if the condition can't be met. The system tries to satisfy the condition in a preference rule, but if for some reason it can't, the system delivers a failure message to the user with Info severity.

The preference rule has this syntax:

```
preference(logic expression, string literal | string variable,
argument, .., argument);
preference(logic expression, string literal | string variable);
preference(logic expression);
```

A preference rule can include an optional explanation message for failure. The message is of `Info` severity, meaning it does not block the user from continuing with the action.

In this example, the preference rule encourages the user to mention the `dBMax` value as `90` and the `requiredKW` value as `500`:

```
type GeneratorSet {
   int requiredKW = [101..10000];
   int dBMax = [0..140];
   preference(dBMax == 90, "90 preferred for dbMax");
   preference(requiredKW == 500,"50 preferred for requiredKW");
}
```

Require Rule

The require rule requires certain components to be included in a relationship when specified conditions are met. Required components can have attributes and quantity specified. The require rule can include an optional explanation message, for the rule failure explanation.

In certain scenarios, you can independently add a type at the header level. This means you can include a specific type even if it isn't explicitly defined as part of any of the relationships you've configured. This capability offers flexibility in managing and including necessary types that might not always fall under a specific relationship structure

Note: When you assign a require rule to a virtual bundle (a bundle related to the sales transaction, where the parent product has no associated price), set one Product Selling Model Option on the required product to Default. For more information on Product Selling Model Options, see Manage Product Selling Model in Revenue Cloud.

The require rule has this syntax:

```
require(logic
expression,relationship[type]{var=value,…,var=value}==integer
value,"Explanation message");
```

In this example, the require rule specifies that if the number of engineers is more than 0, installation is required. The installation will be automatically added upon adding an engineer.

```
type GeneratorSet {
    relation engineers : engineer[0..99];
relation installation : install[0..5];
   require(engineers[engineer] > 0, installation[install],
"Installation is required if engineers are present");
}
```

```
type engineer{}
type install{}
```

Require Rule vs Constraint

In CML, `constraint()` and `require()` can both be used to enforce a certain behavior, but they operate differently. A constraint focuses on the logical consistency, while a require rule focuses on the physical presence of products.

Here's a comparison between `constraint()` and `require()`.

Feature	constraint()	require()
Primary goal	Validates if a condition is met (LHS) and operates on the result (RHS).	Forces a product to be present.
Engine action	Resolves the constraint or displays an error if there are no options to resolve.	Adds the required product to the quote.

Exclude Rule

The exclude rule is used to automatically remove a specific type in a relationship if a certain condition is met.

The exclude rule has this syntax:

```
exclude(logic expression,relationship[type],"Explanation
message");
```

The type must be leaf type, a node without children.

In the exclude rule, if a user sets attribute values in the PCM that violate the rule requirements, the constraint engine overrides the user input in order to validate the constraint. This behavior is different than other constraints, in which the constraint engine doesn't override user input, but displays an error if user input violates the constraint. See How User Input Order Affects Constraint Engine Behavior.

In this example, the exclude rule automatically removes the `Heater_120` heater from the type `GeneratorSet` if the `Voltage3` is greater than or equal to `4160`.

```
type GeneratorSet {
    int Voltage3 = [120..13800];
    relation Heaters : Heater_120 [1..3];
```

```
    exclude(Voltage3 >= 4160, Heaters[Heater_120]);
}
type Heater_120 {}
```

Action Rule

The CML Action Rule is defined using the `rule()` keyword. Its primary purpose is to execute a designated action, specified as a string literal, when a condition is met. This action is typically handled by external systems, such as the Product Configurator API or custom code, to manage business processes, workflows, or complex constraints that fall outside the constraint engine's primary scope.

Action rules have this syntax:

```
rule(condition, action, arg, ..., arg)
rule(<condition>, <action>, "attribute", <attribute>);
rule(<condition>, <action>, "attribute", <attribute>, "value",
[<attribute values>]);
rule(<condition>, <action>, "relation", <relation>, "type",
<type>);
```

`condition` is any logic expression such as a constraint in CML.

`action` is a string literal that specifies an action that can be interpreted by the Product Configurator API. The Product Configurator API supports these actions:

- Hide: hide attribute, attribute value, product option
- Disable: disable attribute, attribute value, product option
- args are a list of arguments needed to execute the action. An argument is a pair including a string literal and an identifier, a literal, or a domain, enclosed in brackets [ ] to specify multiple values. The string literal specifies what kind of argument follows. The identifier attribute can be defined in the type. The engine retrieves the argument value and passes it to the caller to execute the action.

Hide/Disable Rule

The Hide/Disable Rule uses the `rule()` keyword to conditionally remove an element from the selection menu (hide) or preserve it in the menu while preventing user selection (disable). This functionality can be applied:

- On a bundle, hide a component to remove it from the selection menu, or disable a component to preserve it in the menu but prevent users from selecting options for it.

- On an individual product, hide an attribute to remove it from the selection menu, or disable an attribute to preserve it in the menu but prevent users from selecting options for it.
- On an attribute, hide or disable an attribute value to preserve it in the menu but prevent users from selecting options for it. For attribute values, the hide and disable rules have the same behavior.

Note: In Visual Builder in Salesforce, for attribute values, only the hide rule is enabled. When you apply the hide rule to an attribute value in Visual Builder, the value appears in the menu but selections are disabled.

The hide and disable rules use this syntax, where `action` is replaced by either `hide` or `disable`.

```
rule(logic expression, action, actionScope, actionTarget)
rule(logic expression, action, actionScope, actionTarget,
actionClassification, actionValueTarget)
```

Example: Hiding and Disabling Features

In the example in this section, rules rely on specific variables to define the scope and target of the action.

Variable in Rule	Purpose	Example from Generator Set	Description
`logic expression` (Declaration)	Condition upon which the rule occurs.	`requiredKW <= 500`	The logical test that triggers the action.
`action`	Designates whether the rule is hide or disable.	`"hide", "disable"`	Determines if the element is removed from view or made unselectable.
`actionScope`	Designates whether the rule acts on an attribute or relation scope.	`"attribute", "relation"`	Specifies if the target is a variable property or a component relationship.
`actionTarget`	Designates the specific variable or	`"Voltage", "StarterMotors"`	The CML name of the attribute or relation.

	relation that the rule acts on.		
`actionClassification`	Designates whether the rule acts on a type or a value.	`"type", "value"`	Used when targeting components within a relation (type) or specific options within an attribute domain (value).
`actionValueTarget`	Designates the specific type or value that the rule acts on.	`"7976/13800", "StarterMotor_Advanced"`	The specific string value or type name to hide/disable.

```
// --- Component Definitions (For relations referenced below)
---
type LineItem;
type EngineModel : LineItem;
type StarterMotor : LineItem;
type OutputTerminal : LineItem;
type OutputTerminals2HoleLugNEMA : OutputTerminal; // Specific
terminal type

type GeneratorSet : LineItem {
    // Attributes subject to hide/disable rules
    int requiredKW = [101..10000];
    string Voltage = ["220/380", "240/416", "4160/7200",
"7976/13800"]; // Voltage is the attribute being hidden/disabled
    string specialApplication = ["None - Standard", "Motor
Starting"]; // specialApplication attribute contains a value to
hide

    // Relations subject to hide/disable rules
    relation StarterMotors : StarterMotor; // Relation target
(component) to hide/disable
    relation OutputTerminals : OutputTerminal[0..99]; //
Relation being acted upon


    // 1. Disable a Component (Type) in a Relation
```

```
    // If requiredKW is too low (<= 500 kW), the advanced
starter motor component is disabled (visible but unselectable).
    rule(
        requiredKW <= 500,
        "disable",
        "relation",
        "StarterMotors",
        "type",
        "StarterMotor_Advanced"     );

    // 2. Hide an Attribute
    // If the Generator is configured for a special purpose
(Motor Starting), hide the Voltage attribute entirely.
    rule(
        specialApplication == "Motor Starting",
        "hide",
        "attribute",
        "Voltage"
    );

    // 3. Hide a Specific Attribute Value
    // If the power requirement is low (< 2000 kW), hide the
high voltage option (7976/13800) from the Voltage attribute
domain.
    rule(
        requiredKW < 2000,
        "hide",
        "attribute",
        "Voltage",
        "value",
        "7976/13800"
    );

    // 4. Disable a Component Type (Alternate Syntax using the
component name)
    // Disable the OutputTerminals2HoleLugNEMA component if the
required KW is high.
    rule(
        requiredKW >= 5000,
        "disable",
        "relation",
```

```
        "OutputTerminals",
        "type",
        "OutputTerminals2HoleLugNEMA"
    );
}

type StarterMotor_Advanced : StarterMotor; // Subtype used in
the rule
```

Display Product Recommendations

The `recommend` keyword is used within a CML `rule` to display suggestions for related products in the Product configuration interface. The rule defines the condition under which a specific product `type` or `relation` should be suggested to the user.

You can recommend a type, a relation, or both in the same rule. The recommendation rule can be added inside a standalone product, a product bundle, or a virtual container.

Unlike action rules that are interpreted directly by the Product Configurator API, when the condition is met, the engine forces a UI change (hiding or disabling a product option, attribute, or value). Recommendations are not automatically applied to the UI by the configuration engine alone. To suggest types or relations (products/bundles), typically for up-selling or cross-selling based on their current selections at runtime, use the Run Config Rules action within a Salesforce Flow. See Run Config Rules Action in the Revenue Cloud Developer Guide.

- Use an action rule when a selection makes another option invalid or irrelevant. For example, if a user selects a basic warranty, you should "hide" or "disable" the premium support options to prevent a conflicting or impossible setup.
- Use a recommendation rule when you want to nudge the user toward a beneficial add-on. For example, if a user buys a high-end generator, you "recommend" a maintenance service contract. This does not block the user if they choose not to add it.

Here are three examples demonstrating how to implement product recommendation rules.

Example 1: Recommending a Type (Based on Attribute Selection)

This rule is placed within the `GeneratorSet` type. If a user selects an extremely high voltage, the configuration engine recommends a specialized engineer component required for installation or commissioning.

```
type GeneratorSet {
    // Attribute input
    string Voltage = ["277/480", "7976/13800"];
```

```
    // Relation to the component type being recommended
    relation engineers : engineer[0..99];
    // Recommend Engineer Specialist (type) for High Voltage
    rule(
        Voltage == "7976/13800","recommend",
"type", "EngineerSpecialist" product Type
    );
}

// Recommended type
type EngineerSpecialist ;
type engineer;
```

Example 2: Recommending a Relation (Based on Component Quantity)

This rule recommends adding items to an existing relation (`Accessories`) if the user selects a high-end component (such as the most robust enclosure, `Enclosure_SA3`).

```
type GeneratorSet  {
    // Relations defined in the GeneratorSet
    relation Enclosures : Enclosure;
    relation Accessories : Accessory[1..99]; // The relation
being recommended

    // Recommend Accessories when maximum sound dB is chosen
    rule(
        Enclosures[Enclosure_SA3] == 1,
        "recommend",
        "relation",
        "Accessories");
}
type Enclosure ;
type Enclosure_SA3 : Enclosure;
type Accessory ;
```

Example 3: Recommending a Type (From a Virtual/System Container)

This rule is applied at the Quote or System level (using a `virtual` type) and recommends a system integration product if multiple generators are being configured, reflecting the requirement that large projects often need central control components.

```
@(virtual = true)
type Quote {
    // Relation referencing all GeneratorSet instances on the
quote
    @(sourceContextNode =
"SalesTransaction.SalesTransactionItem")
    relation lineItems : GeneratorSet[0..10];

    // Recommend Switchgear for multi-unit orders

    rule(
        lineItems[GeneratorSet] > 1,
        "recommend",
        "type",
        "Switchgear"
    );
}
type GeneratorSet;
type Switchgear;
```

Set Product Selling Model in a Constraint

Use the `productSellingModel` tagname to write a constraint that sets the product selling model (PSM) for a type. You can define a PSM as one time, time-deferred (subscription with end date), or evergreen (recurring subscription with no preset end date). The PSM is updated for new line items at runtime, based on the constraint.

You can't use a CML constraint to update the PSM for an existing quote line. For more information on product selling models, see Manage Product Selling Model in Revenue Cloud in Salesforce Help.

Constraint Example

Using the `GeneratorSet` model, a constraint can be defined that sets the PSM based on a specific operational attribute chosen by the user, such as the `DutyRating`. This assumes that different duty ratings correspond to different billing models (for example, permanent installation versus temporary rental).

This example sets the PSM to a specific ID (for example, `PSM_EVERGREEN_ID`) if the user selects the "Continuous Power (COP)" duty rating.

```
//Global variable PSM ID
define PSM_EVERGREEN_ID "a00Tx000000Qz1g"

type GeneratorSet {
    // Use the productSellingModel tag from the context
definition
@(tagName = "ProductSellingModel")
  string productSellingModel;
    string DutyRating = ["Prime Power (PRP)", "Continuous Power
(COP)", "Emergency Standby Power (ESP)"];
    // Set PSM based on Duty Rating
    constraint(
        DutyRating == 'Continuous Power (COP)' ->
productSellingModel == PSM_EVERGREEN_ID
    );
}
```

Core Concept Examples

Example 1: Use Regex Global Variable

```
// Global Constant: Regex used to parse voltage strings
define VOLTAGE_REGEX "^([11-19]+)/([11-19]+)$"

type LineItem;
type GeneratorSet : LineItem {
   // Core attributes
   @(configurable = false)
   int requiredKW = [101..10000]; // Required power capacity
   string Voltage = ["220/380", "240/416", "277/480",
"7976/13800"];
   string standardsAndCompliance = ["Certification-CSA",
"Listing-UL 2200"];

   // Calculated attributes
   // Surge load is 1.25 times the required KW
   decimal(2) surgeLoadKW = requiredKW * 1.25;
   // Voltage3 extracts the secondary voltage (e.g., 380 or 416)
for checks
```

```
   int Voltage3 = strtoint(regexpreplace(Voltage, VOLTAGE_REGEX,
"$2"), 0);

   // Relation to model components
   relation GeneralModels : GeneralModel;

   // Constraint 1: Ensure model capacity meets the required KW
   constraint(GeneralModels[GeneralModel].powerKW >= requiredKW,
      "Selected Generator Model is insufficient for the required
power.");

   // Constraint 2: UL 2200 standard restricts voltage to 600V
or less (Implication rule)
   constraint(standardsAndCompliance == "Listing-UL 2200" ->
Voltage3 <= 600,
      "The UL 2200 standard covers stationary engine generator
assemblies rated at 600 volts or less.");
}

// Component type definition (GeneralModel is a product
component)
type GeneralModel : LineItem {
   int powerKW = [900, 1200, 1500, 1750, 2500];
   int dB = [20..23];
}
```

Key Technical Details

- The `GeneratorSet` type is related to the `GeneralModel` type through a single relation.

Parent Type	Relation Name	Child Type	Cardinality	Key Attribute Usage
`GeneratorSet`	`GeneralModels`	`GeneralModel`	Not Applicable	Constrained by `requiredKW` on the parent type.

- Parent Type (`GeneratorSet`): Defined as a `LineItem` component, the `GeneratorSet` holds configuration parameters such as the user's power requirement (`requiredKW`), the specified voltage (`Voltage`), and compliance standards

(`standardsAndCompliance`). It also includes calculated attributes like `surgeLoadKW` and `Voltage3`.

- Child Type (`GeneralModel`): Defined as a `LineItem` component, the `GeneralModel` specifies the technical attributes of the selected generator configuration, including its power output (`powerKW`) and noise rating (`dB`).
- Cardinality: The relation `GeneralModels : GeneralModel` specifies the quantity bounds for the component, indicating that exactly 11 instances of the `GeneralModel` type must be included in this configuration.

Example 2: Use Groupby Annotation to Create Virtual Group

Using Groupby annotation for types, this model creates a virtual `VoltageGroup` for every unique voltage (for example, "220/380", "480/277") found in the transaction.

```
// 1. The physical product type
type GeneratorSet {
   @(configurable = false)
   int requiredKW = [101..10000];

   string Voltage = ["220/380", "240/416", "255/440",
"277/480"];

   // Calculation with explicit domain for best practice
   decimal(2) surgeLoadKW = [126.25..12500.00];
   constraint(surgeLoadKW == requiredKW * 1.25);
}

// 2. The virtual container grouped by the "Voltage" attribute
@(split=true, virtual=true, groupBy=Voltage)
type VoltageGroup {
   string Voltage; // The attribute used for grouping
   decimal(2) groupTotalSurgeKW = [0.00..99999.99];

   // Map instances to this group from the transaction line
items
   @(sourceContextNode="SalesTransaction.SalesTransactionItem")
   relation generators : GeneratorSet[0..50];

   // Aggregation: Sum only the surge load of generators in THIS
voltage group
   constraint(groupTotalSurgeKW == generators.sum(surgeLoadKW));
```

```
   // Business Rule: Limit total surge load per voltage branch
   message(groupTotalSurgeKW > 10000, "Warning: Surge load for
Voltage {} exceeds branch capacity!", Voltage);
}

// 3. The top-level system managing the groups
@(virtual = true)
type GeneratorSystem {
   relation generators : GeneratorSet[0..100];
   relation voltageGroups : VoltageGroup[1..10];

   decimal(2) systemTotalKW =
voltageGroups.sum(groupTotalSurgeKW);
}
```

Key Technical Details

- `groupBy=Voltage`: The engine scans all `GeneratorSet` instances and creates one virtual `VoltageGroup` for each unique voltage value detected (for example, one group for all "220/380" units, another for all "277/480" units).
- `sourceContextNode`: This tells the virtual group to pull its "children" (the generators) from the specific path in the Salesforce context where quote line items are stored.
- Bottom-Up Aggregation: Each `VoltageGroup` independently calculates its `groupTotalSurgeKW` based strictly on the generators assigned to it by the `groupBy` logic.
- Explicit Domains: Following best practices, the `surgeLoadKW` and `groupTotalSurgeKW` use explicit domains (for example, `[0.00..99999.99]`) to prevent "NullPointer" or initialization errors during solving.

Example 3: Use Sharingcount Annotation to Reuse Accessory Instances

In this example, we apply sharingcount annotation to the `Accessory` type to allow the engine to reuse accessory instances across the configuration up to a specified limit.

```
// 1. Define the component type with split and sharingCount
// split=true enables parallel solving by splitting quantity
into multiple instances
```

```
// sharingCount=5 allows a single Accessory instance to be
reused 5 times in the model
@(split = true, sharingCount = 5)
type Accessory {
   string category;
   decimal(2) weight;
}

type GeneratorSet {
   @(configurable = false)
   int requiredKW = [101..10000];

   // 2. Define the relationship with the sharing annotation
   // @(sharing = true) allows the engine to satisfy this
relation by
   // "pointing" to existing instances instead of creating new
ones
   @(sharing = true)
   relation Accessories : Accessory[1..99];
}
```

Key Technical Details

- Parallel Solving: By setting `split = true`, the engine can process the 99 possible accessories in parallel rather than sequentially, which is critical for large-scale generator configurations.
- Resource Management: The `sharingCount = 5` tells the solver that it doesn't need to instantiate 99 unique `Accessory` objects; it can reuse the same object definition up to five times across different parts of the configuration graph.
- Relationship Requirement: For `sharingCount` to take effect, the Relation (the port) must also be annotated with `@(sharing = true)` to grant the engine permission to reuse instances.

Example 4: Use contextPath and tagName Annotations

```
// 1. GLOBAL EXTERN DECLARATIONS
// Unifying contextPath (header field) and tagName (context
identifier)
@(contextPath = "SalesTransaction.ShippingCountry", tagName =
"Region_Identifier")
extern string ShippingCountry = "International";
```

```
@(contextPath = "SalesTransaction.ProjectUrgency", tagName =
"Priority_Level")
extern string ProjectUrgency = "Standard";

// 2. PHYSICAL PRODUCT TYPE
type GeneratorSet {
   // Core technical attributes
   @(configurable = false)
   int requiredKW = [101..10000];

   string DutyRating = ["Prime Power (PRP)", "Emergency Standby
Power (ESP)"];

   // Technical calculation with explicit domain (Best Practice)
   decimal(2) surgeLoadKW = [126.25..12500.00];
   constraint(surgeLoadKW == requiredKW * 1.25);

   // 3. LOGIC INTEGRATING EXTERNAL DATA
   // Using contextPath/tagName data to drive technical warnings
   message(ShippingCountry == "US",
           "Regional Notice: Generator must comply with
US-specific UL 2200 standards.");

   message(ProjectUrgency == "Critical" && requiredKW > 5000,
           "High Priority Alert: Large scale power requirement
in a Critical project requires site inspection.",
           requiredKW, "Warning");
}
```

Key Technical Details

- External Variable (`extern`): These are declared outside of any type to hold values supplied by the environment (Salesforce).
- `contextPath` Annotation: This maps the variable directly to a header-level field in the Sales Transaction.
- `tagName` Annotation: This links the variable to a specific Context Tag identifier within the Salesforce Context Definition.

Note: Rules depending on these external variables (like the `ShippingCountry` message) only re-evaluate when a Line Item change occurs (e.g., updating `requiredKW`), not solely when the header field changes.

Example 5: Use Format Specifiers (%s, %d) and Dates in Constraints

In this example, we define a generator configuration that validates the required power (`requiredKW`) against the duty rating and voltage, providing specific feedback when a mismatch occurs.

```
type GeneratorSet  {
   // 1. Core Technical Attributes with explicit domains
   @(configurable = true)
   int requiredKW = [101..10000];

   string DutyRating = ["Prime Power (PRP)", "Continuous Power
(COP)",
                        "Data Center Continuous (DCC)",
"Emergency Standby Power (ESP)"];

   string Voltage = ["220/380", "277/480", "347/600",
"2400/4160"];

   string standardsAndCompliance = ["Certification-CSA",
"Listing-UL 2200"];

   // Date Attribute Definition
   // Date variables represent a specific day.
   // You can define a fixed domain for a date as a range
between two dates.
   date requestedDeliveryDate = ["2024-01-01", "2025-12-31"];

   // Message Rule with Multiple Arguments using placeholders
   message(requiredKW > 5000 && DutyRating == "Emergency Standby
Power (ESP)",
           "High Capacity Alert: The %s rating for %d kW
requires an additional cooling system.",
           DutyRating, requiredKW, "Warning");

   // Constraint with Multiple Arguments using placeholders
```

```
   constraint(Voltage == "2400/4160" && standardsAndCompliance
== "Listing-UL 2200",
              "Configuration Error: Voltage %s cannot be used
with %s standards due to safety limits.",
              Voltage, standardsAndCompliance);

   // Dates can be used in logical expressions with comparison
operators like < or >=.
   message(requestedDeliveryDate < "2024-09-01" && requiredKW >
7000,
           "Schedule Warning: Delivery for %d kW units on %s may
require expedited manufacturing.",
           requiredKW, requestedDeliveryDate, "Info");

   // You can also use date annotations to make a constraint
active only during a specific period.
   @(startDate="2024-11-01", endDate="2024-12-31")
   constraint(requiredKW > 8000 -> standardsAndCompliance ==
"Certification-CSA",
              "End-of-Year Requirement: High-capacity units
ordered in Q4 2024 must meet CSA Certification.");
}
```

Key Technical Details

- Multiple Arguments: You can pass several variables after the string literal. The engine maps them to the placeholders in the exact order they appear.
- Format Specifiers
    - `%s` is used for string variables (for example, `DutyRating` or `Voltage`).
    - `%d` is used for integer variables (for example, `requiredKW`).
- Placeholder Usage: Alternatively, you can use the {} syntax, which the constraint engine automatically replaces with the actual argument values during runtime evaluation.
- Comparison Logic with Dates: You can treat dates as comparable values in `constraint` or `message` rules to enforce scheduling logic (for example, ensuring a delivery date is not too early for a complex build).
- Date Annotations: Unlike standard variables, constraints themselves can be "date-aware" using the `@startDate` and `@endDate` annotations. This allows you to enforce certain business rules (such as seasonal compliance or promotional requirements) only during a specific window of time.

Example 6: Use Arithmetic Calculations and Functions

This example demonstrates using aggregate functions (a virtual type for transaction-level aggregation) and mathematical functions within the `GeneratorSet` type for precise quantity calculation.

```
// --- Component Definition —
type GeneratorSet  {
    int requiredKW = [101..10000];
    int quantity = [1..10];
    // BEST PRACTICE: Define derived attributes with an explicit
domain
    decimal(2) surgeLoadKW = [0.00..12500.00];
    // BEST PRACTICE: Put calculations inside of constraints
    constraint(surgeLoadKW == requiredKW * 1.25);
}
// --- Accessory Definition ---
type Accessory  {
    int weight_kg = [1..100];
}
// --- Virtual System Type (Aggregation Context) ---
@(virtual = true)
type System {
    @(sourceContextNode =
"SalesTransaction.SalesTransactionItem")
    relation generators : GeneratorSet[0..10];
     relation shipmentbatch : ShipmentBatch [0..10];
    // BEST PRACTICE: Attributes must have explicit domains
    decimal(2) totalQuotedLoad = [0.00..125000.00];
    int highPowerCount = [0..10];
    // Aggregate calculations in separate constraints
    constraint(totalQuotedLoad == generators.sum(surgeLoadKW));
    // VARIATION: Condition-based count
    // count() requires a logical expression
    constraint(highPowerCount == generators.count(requiredKW >
5000));
    // Ensuring Shipment batch crates are similar to the number
of generators
    message(
        shipmentbatch.requiredCrates !=
generators[GeneratorSet],
```

```
        "The Shipment items ({}) must match the number of
Generators ({})",
        shipmentbatch.requiredCrates, generators[GeneratorSet],
        "Error"
    );
}
type ShipmentBatch {
    int totalItems = [1..100];
    int itemsPerCrate = 10;
    // BEST PRACTICE: Define derived attributes for the relation
in the parent type
    int totalWeight = [0..1000];
    int accessoryCount = [0..10];
    // Aggregates within the relation block
    relation accessories : Accessory[0..10] {
        accessoryWeight = sum(weight_kg);
        accessoryCount = count(weight_kg > 0);
    }
    // Mapping relation attributes to parent variables via
constraints
    constraint(totalWeight == accessories.accessoryWeight);
    constraint(accessoryCount == accessories.accessoryCount);
    int requiredCrates = [0..100];
    // Mathematical Function Example: CEIL
    constraint calculateCeil(requiredCrates == ceil(totalItems /
itemsPerCrate));
}
```

Key Considerations for Calculations and Aggregations

When implementing these functions in CML, follow these architectural best practices for robustness and performance.

1. Explicit domains are required: All the derived attributes that result from a calculation or aggregation must have an explicit variable domain definition. This practice ensures accurate aggregation and helps prevent runtime errors.
2. Separate calculation from declaration: Define the aggregation or calculation in a separate constraint rather than using an inline derived attribute declaration (for example, `int total = items.sumPrice;`). This separation helps avoid issues where domains may not initialize correctly.

3. Correct Pattern: Define the relation aggregate (`totalQty = sum(quantity);`) and then enforce the result via a constraint (`constraint(totalItemCount == items.totalQty);`).

CML Best Practices

To prevent performance degradation or unexpected behaviors when the constraint engine executes CML code, follow these practices when writing code. For tips on troubleshooting, see Debugging CML.

1. Relationship Cardinality: Specify the Smallest Range Required

In a relationship, cardinality is the quantity of instances of the same type. Specify the smallest required cardinality for a variable, to avoid testing unneeded combinations of values. If you specify a higher cardinality than required, or don't specify cardinality, the constraint engine tests more combinations, which impacts performance.

This example doesn't specify cardinality. The constraint engine tries to set a quantity with 1, 2, 3, all the way up to 9,999:

```
relation engine : Engine;
```

This example specifies minimum and maximum cardinality as 0 and 1, so the constraint engine sets the quantity to 1. The engine tests fewer combinations to find a solution.

```
relation engine : Engine[0..1];
```

2. Decimals and Doubles: Consider the Impact of Scale on Performance

In a decimal or double, scale is the number of digits that follow the decimal point. Using decimals and doubles in expressions can cause performance problems due to the number of permutations.

In this example, myNumber is a double with a scale of 2. The value can be 0.00, 0.01, 0.02, all the way up to 2.99, which can impact constraint engine performance:

```
double(2) myNumber = [0..3];
```

In this example, myNumber is an integer. The value can only be 0, 1, 2 or 3, which has less impact on constraint engine performance:

```
int myNumber = [0..3];
```

3. Variable Domains: Keep Domains as Small as Possible

A variable domain is the set of all possible values that the variable can take. In this example, the variable color has a domain with three values:

```
string color = [“Red”, “Yellow”, “Green”];
```

The larger the domain, the more possible values for the variable, which means more combinations for the engine to test. A large domain can impact performance and lead to slower searches, errors, or unexpected behaviors.

4. Calculating Values: Put Calculations Inside of Constraints

To calculate a value, put the calculation inside of a constraint, instead of in an inline expression.

For example, to calculate area, use this constraint:

```
constraint(area == length * width)
```

Avoid this example, which calculates the area with an inline expression, and can impact performance:

```
area = length * width.
```

5. Relationships: Combine Relationships to Reduce Performance Impact

Creating multiple relationships on a type can impact performance. When possible, combine relationships to improve performance.

When possible, avoid this example, which includes separate relationships for Mouse and Keyboard, two accessories in a product bundle:

```
relation mouse : Mouse;
relation keyboard : Keyboard;
```

Follow this example, which uses one relationship for Accessories, which can include Mouse, Keyboard, and other accessories.

```
relation accessories : Accessories;
```

6. Sequence: Use the Sequence Variable Annotation to Specify the Order of Execution

If a constraint model includes multiple attributes and relationships that should follow a certain order of execution, use the sequence variable annotation to specify the order. The constraint engine follows sequence designations in satisfying constraint requirements and resolving constraint violations.

In this example, for the Desktop type, the sequence annotation directs the constraint engine to set the default values for attributes in this order:

- Display: sequence=1
- Windows_Processor: sequence=2
- Display_Size: sequence=3

```
type Desktop {
    @(defaultValue = "1080p Built-in Display", sequence=1)
    string Display = ["1080p Built-in Display", "4k Built-in
Display", "2k Built-in Display"];

    @(defaultValue = "15 Inch", sequence=3)
    string Display_Size = ["15 Inch", "24 Inch", "13 Inch", "27
Inch"];

    @(defaultValue = "i5-CPU 4.4GHz", sequence=2)
    string Windows_Processor = ["i5-CPU 4.4GHz", "i7-CPU
4.7GHz", "Intel Core i9 5.2 GHz"];

    constraint(Display == "1080p Built-in Display" &&
Display_Size == "15 Inch" -> Windows_Processor == "i7-CPU
4.7GHz");
}
```

For Desktop, Display is set to 1080p, Windows_Processor to i5-CPU, and Display_Size to 15 Inch.

The constraint specifies that a type with Display of 1080p and Display_Size of 15 Inch must have a Windows_Processor of i7-CPU:

```
constraint(Display == "1080p Built-in Display" && Display_Size
== "15 Inch" -> Windows_Processor == "i7-CPU 4.7GHz");
```

The Windows_Processor default value of i5-CPU for Desktop violates the constraint. In order to satisfy the constraint and resolve the violation, the constraint engine uses a different Display_Size for Desktop, such as 24 Inch.

If the user manually updates Display_Size for Desktop to 15 Inch in the Product Configurator, the constraint engine updates Windows_Processor to i7-CPU to satisfy the constraint.

Sequence and Configurable

Using the `configurable` property with the `sequence` variable affects how the solver handles attributes. When `configurable` is set to `true`, the user can set attribute values, and the solver doesn't override user input unless no other solution exists. When `configurable` is set to `false`, the solver sets attribute values.

In this example for type A, `attribute1` is set to `configurable = true`. The user can set the value, and the solver doesn't override the user input unless no other solution exists. When the left-hand side of the rule is true, the constraint doesn't change `attribute1` to `"Enum3"`.

```
type A : System {
    @(defaultValue = "Enum1", domainComputation = "true",
configurable = true, sequence = 48)
    string attribute1 = ["Enum1", "Enum2", "Enum3"];

  @(configurable = false, defaultValue = "0", sequence = 30)
    decimal(2) attribute2 = f(x, y, z);
    constraint((attribute2 > 3 && attribute2 <= 5) -> attribute1
== "Enum3", "message");
}
```

In this example for type B, `attribute1` is set to `configurable = false`. The solver propagates values to satisfy constraints. When the left-hand side of the rule is true, the constraint automatically sets `attribute1` to `"Enum3"`.

```
type B : System {
    @(defaultValue = "Enum1", domainComputation = "true",
configurable = false, sequence = 48)
    string attribute1 = ["Enum1", "Enum2", "Enum3"];

  @(configurable = false, defaultValue = "0", sequence = 30)
    decimal(2) attribute2 = f(x, y, z);
    constraint((attribute2 > 3 && attribute2 <= 5) -> attribute1
== "Enum3", "message");
}
```

Use mindful sequencing in CML to avoid backtracking by the solver when looking for a solution, as in this example:

```
type LaptopProBundle {

    //relation mouse : Mouse[1..20];
    //The relation mouse has the highest sequence
    relation warranty : Warranty[0..10];
    relation software : Software;
    relation printerBundle : PrinterBundle;
    relation laptop : Laptop[1..10];
    relation mouse : Mouse[1..20];
    //Put highest sequence last to avoid backtracking

    int mouseQty = laptop[Laptop] + warranty[Warranty];

    constraint(mouse[Mouse] > 0 -> mouse[Mouse] == mouseQty,
                "mouse[Mouse] = laptop[Laptop] +
warranty[Warranty]");
```

7. Automatically Add a Product: Define as a Separate Constraint

If you need to automatically add a product, and also set attributes on the product, define these procedures as separate constraints, as in this example:

```
constraint(laptop[Laptop] > 0, warranty[Warranty] > 0);
constraint(warranty[Warranty] > 0, warranty[Warranty].type ==
"Premium");
```

Avoid this example, which automatically adds a product and sets attributes on the product, in the same constraint:

```
constraint(laptop[Laptop] > 0, warranty[Warranty] > 0 &&
warranty[Warranty].type == "Premium");
```

8. Access Quantity in CML: Cardinality and Attribute Constraints

There are two ways to access quantity in CML:
A **cardinality constraint** creates or validates the presence of components. The constraint engine adds or removes instances to satisfy the conditions of the constraint.
An **attribute constrain**t, such as `lineItemQuantity` or `ItemEndQuantity`, only reads or validates. The constraint engine validates the expression, but doesn't configure to satisfy the conditions of the constraint

For best performance, follow these guidelines:

- Use a cardinality constraint whenever possible. Use `lineItemQuantity` or `ItemEndQuantity` only when a cardinality constraint can't meet the business need.
- Treat `lineItemQuantity` and `ItemEndQuantity` as read-only. Use only in calculation or evaluation rules.
- Keep scope in mind. When a product can have multiple instances, read `lineItemQuantity` or `ItemEndQuantity` per instance. Avoid reading the attribute at a parent or aggregate scope where it can be unbound or ambiguous.
- Don't use `lineItemQuantity` or `ItemEndQuantity` to create components. Drive component creation by cardinality, not by attribute references.

Use constraint patterns similar to these examples.

Do this:

```
constraint(mouse[Mouse] == warranty[Warranty])
```

Avoid this:

```
constraint(mouse[Mouse].lineItemQuantity ==
warranty[Warranty].lineItemQuantity)
```

Do this:

```
constraint(mouse[Mouse] == 3)
```

Avoid this:

```
constraint(mouse[Mouse].lineItemQuantity == 3)
```

9. Pricing Fields Not Supported in CML

Pricing fields, such as ListPrice, NetUnitPrice, and others, are not supported in CML and should not be used in constraint models. CML is designed to enforce configuration logic for products, not to perform pricing calculations. Attempting to reference or manipulate pricing fields in CML code leads to errors and unexpected behaviors in the constraint engine. Use dedicated pricing or calculation mechanisms outside of the CML constraint model for such functionality.

Business-Centric CML Guidelines: Quantity and Aggregation Functions

Business Scenario

CML must accurately calculate the total sum or aggregate of specific attributes like quantity or userCount across child components, especially in complex configurations requiring group-level aggregation.

The main modeling obstacles when performing aggregation in CML involve:

- Initialization Errors: Preventing runtime errors, such as `NullPointerException`, which can occur if derived aggregate attributes lack explicit domains.
- Circular Dependencies: Avoiding calculation loops where the parent and children mutually rely on aggregated totals, often involving the `total()` function. If these loops are not broken, the aggregated variable becomes "not bound", which causes the solution to fail.

User Workflow

As a sales representative, when I am configuring a bundle product in the Configurator window, I modify the quantities or specific attributes of the individual child components. I expect the constraint engine to immediately and accurately calculate the overall aggregated totals for the parent product, such as the totalItemCount or sumOfUsers.

Business-Centric CML Examples

These CML structures implement quantity aggregation and resolve calculation dependencies.

Example 1: Derived Aggregates (Total Quantity or Sum)

This pattern ensures correct initialization and binding of calculated parent totals by using an explicit domain and a separate constraint.

```
type Bundle {
  int totalItemCount = [0..100]; // Explicit domain
  // Aggregate quantity
    relation items: LineItem[0..10] {
    totalQty = sum(quantity);
  }
```

```
  constraint(totalItemCount == items.totalQty); // Separate
constraint
}
type LineItem{
    int quantity = [0..10];
}
```

Example 2: Resolving Circular Dependencies

This pattern breaks unsolvable loops by enforcing unidirectional data flow (Parent dictates Child quantity based on Child aggregates).

```
type LineItem {
    int weight = [1..100];
}

type ParentBundle {
    // Explicit domain ensures the solver initializes correctly
for calculations
    decimal(2) totalDistributionValue = [0.00..1000.00];

    // RELATION: THE "READ" PHASE
    // Data flows UP from children to the parent via the port
attribute
    relation children: LineItem[1..10] {
        sumOfChildWeights = total(weight);
    }

    // CONSTRAINT 1: THE "CALCULATION" PHASE
    // @(sequence = 1) ensures the engine reads child data and
calculates first
    @(sequence = 1)
    constraint calcTotalDistribution (
        totalDistributionValue == children.sumOfChildWeights /
100
    );

    // CONSTRAINT 2: THE "WRITE/ENFORCE" PHASE
    // @(sequence = 2) ensures this happens last to push values
DOWN to children
    @(sequence = 2)
    constraint setChildQuantity (
```

```
        children[LineItem] == totalDistributionValue * 2
    );
}
```

Example 3: Grouped Aggregation (Sum of Users Across Regions)

This advanced pattern uses the `@(groupBy=attribute)` annotation to create virtual components (RegionGroup) for aggregation, referencing the source data relation in the parent type (SubscriptionOrder.licenses) using the sourceContextNode annotation.

```
// 1. Base Product Type (e.g., Regional License)
type RegionalLicense {
  int regionId = [0..100]; // Attribute for grouping
  int userCount = [0..100];
}

// 2. Virtual Group Type (Performs grouping and user sum)
@(split=true, virtual=true, groupBy=regionId)
type RegionGroup {
  int regionId;
  int groupTotalUsers; // Sum of userCount for this region

  // Relation to licenses matching this region group
  @(sourceContextNode="SubscriptionOrder.licenses")
  relation licenses: RegionalLicense[0..100];

  // Constraint: Calculate the aggregate user count within this
group
  constraint(groupTotalUsers == licenses.sum(userCount));
}

// 3. Parent Order Management (Aggregates all region totals)
@(virtual=true)
type SubscriptionOrder {
  // Overall total users is calculated by referencing the group
totals
  int totalUsers = regionGroups.groupTotalUsers;

  relation licenses: RegionalLicense[0..500];
```

```
  relation regionGroups: RegionGroup[1..10];
}
```

Debugging CML

To debug constraint models and troubleshoot performance issues, enable debug logging in Apex and set the debug log level to FINE. For more information on debug logging in Salesforce, see these topics in Salesforce Help:

- Set Up Debug Logging
- Debug Log
- Debug Log Levels

Use the Apex log to get information about configurator engine performance when running a constraint model, including performance degradation or unexpected behavior. To improve performance, modify the constraint model based on information in the log.

For tips on writing trouble-free CML, see CML Best Practices.

About the Apex Debugging Log File

The Apex debugging log file contains three sections:

RLM_CONFIGURATOR_BEGIN

JSON representation of the request payload to ExecuteConstraintsRESTService:

```
 "contextProperties" : { },
  "rootLineItems" : [ {
    "attributes" : { },
    "properties" : {  },
    "ruleActions" : null,
    "attributeDomains" : { },
    "portDomainsToHide" : { },
    "lineItems" : [ {} ]
   } ],
  "orgId" : "00Dxx0000006H2F"
}
```

RLM_CONFIGURATOR_STATS

Key statistics of the request execution by the constraint engine, as in this example:

```
"rootId" : "0QLxx0000004D1uGAE",
      //Root ID that is being configured
  "Product" : "SFDC License",
      //Root product name
  "Total Execution Time" : "2ms",
      //Total solver time
```

```
  "Constraints Execution Stats" : "Distinct: 18 Total: 70",
      //Number of distinct and total constraint satisfaction attempts
  "Solving goal AndGoal([ConfigureComponentGoal(RootProduct
RootProduct_0)]) took " : "2ms",
      //Total solver time for the goal
  "Configurator Stats" : "Total Time 2ms",
      //Total time
  "Number of Component" : "6",
      //Number of components instantiated
  "Number of Variables" : "42",
      //Number of variables instantiated
  "Number of Constraints" : "13",
      //Number of constraints instantiated
  "Number of Backtracks" : "0",
      //Number of backtracks solver did for the last choice point
  "Constraints Violation Stats" : "Distinct: 0 Total: 0",
      //Distinct and total number of constraint violations followed by a list of top 10
  "ChoicePoint Backtracking Stats" : "Distinct: 0 Total: 0"
      //  Distinct and total number of backtracked choice points followed by a list of
      //  top 10
} ]
```

RLM_CONFIGURATOR_END

JSON representation of the response payload from ExecuteConstraintsRESTService:

```
"id" : "0QLxx0000004D1uGAE",
  "rootId" : null,
  "parentId" : null,
  "cfgStatus" : "User",
  "name" : "RootProduct",
  "relation" : null,
  "source" : "SalesTransaction.SalesTransactionItem",
  "qty" : 1,
  "actionCode" : null,
  "modelName" : "Support_instance_variable_in_CML",
  "productId" : "01txx0000006iP2AAI",
  "productRelatedComponentId" : null,
  "attributes" : {},
  "properties" : {},
  "ruleActions" : [ {} ],
   "attributeDomains" : {},
  "portDomainsToHide" : {},
  "lineItems" : [ {} ]
```

```
} ]
```

Use the Apex Debugging Log File

To find possible reasons for the performance problems and identify solutions, look at the RLM_CONFIGURATOR_STATS section of the log file. See the values for Total Execution Time, Constraints Violation Stats, and ChoicePoint Backtracking Stats.

For example, consider how the constraint engine performs with this sample constraint model. In the constraint model, the value of the `volts` variable is greater than 110/10000 (`volts = power/amps * 9999;`). The constraint engine must backtrack the `power` variable to find a value that satisfies the constraint, starting with 0.01, 0.02, and so on until it reaches a valid value.

```
relation laptops : Laptop[1..9999];

    @(sequence = 1)
    decimal(2) power = [0..500];

    @(sequence = 1)
    int amps = [1..5];

    decimal(2)  volts = (power / amps) * laptops[Laptop];

    constraint(volts > 110);
```

In the log file for this constraint model, see the execution statistics for Total Execution Time, Constraints Violation Stats, and ChoicePoint Backtracking Stats:

```
"rootId" : "ref_a67c6632_fa1f_40b4_8093_226a9ab8a4d0",
"Product" : "Laptop",
"Total Execution Time" : "676ms",
"Constraints Execution Stats" : "Distinct: 2 Total: 132006",
"Solving goal AndGoal([ConfigureComponentGoal(Laptop Laptop_0)])
took " : "677ms",
"Configurator Stats" : "Total Time 677ms",
"Number of Component" : "1",
"Number of Variables" : "4",
"Number of Constraints" : "1",
"Number of Backtracks" : "49500",
"Constraints Violation Stats" : "Distinct: 1 Total: 41250",
```

```
"IntComparison(GT,[DecimalVar(volts)])" : "41250",
"ChoicePoint Backtracking Stats" : "Distinct: 2 Total: 98999",
"VariableChoicePoint(DecimalVar(power))" : "49500",
"VariableChoicePoint(IntVar(amps))" : "49499"
```

Optimally, execution time for a constraint model is less than 100 milliseconds, with fewer than 1,000 backtracks and no violations. Values for the constraint model example are significantly higher, indicating that the constraint engine is performing inefficiently. To improve performance in this example, reduce the domain of the `power` variable without reducing the solution space. For example, define the domain as `[110..500]` instead of `[0..500]`. This change reduces the number of backtracks the constraint engine performs to find a solution.