Constraint Modeling Language (CML) User Guide

Version 1.6
Winter ‘25

What Is CML?

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, and enforce business logics 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* with fixed values that set up the foundation of the constraint model.
- 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.
- 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. For more information on context tags, see Create a Context Definition in Salesforce Help.
- 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 in order for the constraint on the laptop to run.

For more information on constraint models, see Constraint Model Example: Modeling a House and CML Core Concepts.

Constraint Model Example: Modeling a House

The Costraint Model for a House example uses CML to define a simple house design. Each element in the model corresponds to a core CML concept. For more information on each concept, see the topics linked here.

Global Properties and Settings: In the example, `COLORS` and `MAX-ROOM` are global constants whose values remain fixed throughout the model.
Types: `House`, `Room`, `LivingRoom`, and `Bedroom` are types that represent entities. `LivingRoom` and `Bedroom` inherit from the `Room` type. `MasterBedroom` inherits from `Bedroom`.
Variables: The `House` type defines several variables including `address`, `numberOfRooms`, and `totalArea`. The `Room` type defines several variables including `width`, `length`, `area`, and `color`. The color for each room is chosen from the values in the global constant `COLORS`.
Relationships: The `rooms` relationship connects the `House` type to two `Room` types, `LivingRoom` and `Bedroom`. Minimum and maximum cardinality 1 and `MAX_ROOM` (`Room[1…MAX_ROOM]`) enforces the rule that each house must have between 1 and 10 rooms where the global constant `MAX_ROOM` has a value of 10.
Constraints: The constraint `rooms[Bedroom] > 0` enforces the rule that each house must have at least one bedroom (`>0`). The constraint `numberOfRooms == rooms[Room]` enforces the rule that the number of rooms in each house must be equal to the value of the variable `numberOfRooms`. This forces the configuration engine to create numberOfRooms in the configuration. (`order (LivingRoom, Bedroom)` instructs the configuration engine to create LivingRoom first and BedRoom second.

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

Constraint Model for a House

```
define COLORS ["Red", "Blue", "White"]
define MAX_ROOM   10
define MAX_ROOM_SIZE 100

type House {
   string address;
   int numberOfRooms = [1..MAX_ROOM];
   decimal(2) totalArea = rooms.sum(area);

   relation rooms : Room[1..MAX_ROOM] order (LivingRoom,
Bedroom);

   // Ensure there is at least one bed bedroom
   constraint(rooms[Bedroom] > 0);
```

```
   constraint(numberOfRooms == rooms[Room]);

   // Show a message if the area is very large
   message(totalArea > 5000, "This is a spacious house!");
}

type Room {
   decimal(2) width = [1..MAX_ROOM_SIZE];
   decimal(2) length = [1..MAX_ROOM_SIZE];
   decimal(2) area = width * length;
   string color = COLORS;

   // retrieve total area from the house
   decimal(2) houseArea = parent(totalArea);
}

type LivingRoom : Room;
type Bedroom : Room;
Type MasterBedroom : Bedroom;
```

CML Core Concepts

See these topics for information on each core concept and the ways they work together:
Global Properties and Settings
Types
Variables
Relationships
Constraints

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 can be referenced multiple times.

In this example, `MAX_ROOM_SIZE` is a constant that restricts dimensions of a room and `COLORS` is a list that can be referenced to assign colors to different rooms.

```
define MAX_ROOM_SIZE 1000
define COLORS ["Red", "Blue", "Green", "White"]

// define max double model can support
property maxDouble = 99999999;

type House {
    string color = COLORS;
}

type Room {
    int size = [0..MAX_ROOM_SIZE];
    string color = COLORS;
}
```

External Variables

External variables are global CML variables that are defined outside of any CML type. The values for external variables are usually set by the environment that launches the constraint solver engine. If the external variable is not mapped to any external data source, its value is set to the default value. External variables can be used to import sales transaction fields from the context definition. Use the `contextPath` annotation to denote sales transaction fields. See External Variable Annotations.

Basic Declaration Syntax

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

This example shows external variables of integer and decimal data types with default values. Their values are set to the values of the same name tag in the context header.

Example 1: Using External Variables with Constraints

```
// External variable declarations
     extern int MAX_BEDROOMS = 10;
     extern decimal(2) MIN_AREA = 500.00;
     extern decimal(2) PRICE_PER_SQFT = 250.00;

     type House {
         string ownerName;
         decimal(2) areaInSquareFeet;
         int numberOfBedrooms;
         decimal(2) estimatedPrice = areaInSquareFeet *
     PRICE_PER_SQFT;

         // Constraints using external variables
         constraint(areaInSquareFeet >= MIN_AREA);
         constraint(numberOfBedrooms <= MAX_BEDROOMS);
     }
```

Example 2: Using External Variables with Context Path Annotation

```
// External variable declaration with context path annotation
     @(contextPath = "SalesTransaction.TotalAmount")
```

```
extern decimal TotalAmount;

type House {

    @(tagName = "HouseFeature,HouseFeature")
    string houseFeatureField;

    string msg1 = "House feature is " + featureField;
    string msg2 = "Total price is " + TotalAmount;

    message(true, msg1);
    message(true, msg2);
}
```

Data Types for External Variables

Data Type	Description
boolean	A value that can only be assigned true, false, or null
date	A value that indicates a particular day. Same as java local date
double(n)	A 64-bit number that includes a decimal point. It is same as double in Java
decimal(n)	A fixed-point numeric value with n decimal places
int	Integer. A 32-bit number that doesn't include a decimal point, same as int in Java
string	Any set of characters surrounded by single quotes

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, like account name, sales transaction total, or address, directly from their context definition.

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.

Type Declaration Example

In this example, `House` is a type with the variables `address` and `numberOfRooms`.

```
type House {
   string address;
   int numberOfRooms;
}
```

In this example, `House` is a type with a relationship to `rooms`.

```
type House {
   relation rooms : Room[1..5] order (LivingRoom, Bedroom);
}

type Room;
type LivingRoom : Room;
type Bedroom : Room;
```

Type Hierarchies

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

In this example, `Bedroom` and `LivingRoom` inherit from `Room,` and `MasterBedroom` inherits from `Bedroom`, allowing the types to share common variables and relationships.

```
type Room;
type Bedroom : Room;
type MasterBedroom : Bedroom;
type LivingRoom : Room;
```

Type Annotations

You can annotate types to add information.

In this example, the annotation `split = true` specifies that the quantity of the instance of type `Building` is always 1.

```
@(split = true)
type Building;
```

Type Annotations

Annotation	Possible Values	Description
groupBy	Variable name	Used with `virtual = true,` specifies that the engine needs to group the instances of this child type in its relationship by the attribute value specified by this annotation, and create a virtual container to hold its child type instances.
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 `split=none` (the default), there are multiple instances of the same type in the relationship, with different quantities.
virtual	true, false	If `true`, specifies the indicated type to be the type for the transaction header (such as Quote or Order).

Variables

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

Variable Data Types

Variables support multiple data types including boolean, date, decimal, and so on.

In this example, for the `House` type:

- The variable `ownerName` is a string
- The variable `areaInSquareFeet` is a decimal value with a scale of 2

```
type House {
   string ownerName;
   decimal(2) areaInSquareFeetarea;
}
```

Data Types for Variables

Data Type	Description
boolean	A value that can only be assigned `true`, `false`, or `null`.
date	A value that indicates a particular day, the same as local date in Java.
double(n)	A 64-bit number that includes a decimal point, the same as double in Java.
int	Integer. A 32-bit number that doesn't include

	a decimal point, the same as int in Java.
string	Any set of characters surrounded by single quotes.
string[]	Used in multi-select picklists to allow the user to select more than 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.

Variable Domains and Domain Restrictions

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

This example defines a `Room` type with five variables, each having a fixed domain:

- `color` can be one of the specified string values
- `height` can be one of the specified integer values
- `width` can be any integer between 2 and 10 (inclusive)
- `depth` can be an integer in the range 3-5, exactly 7, 9 -11, or 13-15
- `area` must be exactly 300

```
type Room {
   string color = ["Red", "White", "Blue"];
   int height = [1, 3, 5, 7];
   int width = [2..10];
   int depth = [3..5, 7, 9..11, 13..15];
   int area = 300;
}
```

Variable Functions

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

In this example, the `totalHeight` variable calculates the sum of all height values from the descendants of the `House` type.

```
type House {
   decimal(2) totalHeight = sum(height);
}
```

This example shows how functions can be used both within a constraint and in variable declarations.

```
define MAX_ROOM 10
define MAX_ROOM_SIZE 100
type House {
  relation rooms : Room[0..10];
  decimal(2) totalWidth = rooms.sum(width);
  decimal(2) minWidth = rooms.min(width);
  decimal(2) maxWidth = rooms.max(width);
  decimal(2) redRoomCount = rooms.count(color == “Red”);
  constraint(rooms.sum(width) > 0); // Sum Example
  preference(rooms.count(color == "Red") > 5); //Count Example
  constraint(rooms.count(color == “Red”) > 2 &&
rooms.count(color == “Blue”)); //Count example showing how to
count multiple conditions
  constraint(rooms.min(width) > 0); // Min example
  constraint(rooms.max(length) == 100; // Max example
}
type Room {
   string color = ["Red", "Blue", "Green"];
   decimal(2) width = [1..MAX_ROOM_SIZE];
   decimal(2) length = [1..MAX_ROOM_SIZE];
}
```

Proxy Variables

Use proxy variables to reference the variables of related types, including parent, root, and sibling types.

These proxy variables are supported. For more information, see each linked topic.
Cardinality
Parent
this.quantity

Cardinality

The `cardinality` proxy variable refers to the cardinality of a relationship (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>, <port name>)
```

Each parameter value can be a string or a string variable. If the port name isn't specified, the engine searches all ports to find the type. Use the `cardinality` proxy variable without the port name parameter to get cardinality dynamically using another string variable, or to reference a product that isn't defined in CML.

In this example, the `cardinality` proxy variable specifies the quantity of all instances of rooms of type `Bedroom` and `Bathroom`.

```
type House {
  string roomSelection = ["Bedroom", "Bathroom"];

  int roomCardinality = cardinality(roomSelection);

  // alternatively, you can use the string representation of
//a type name; for example
   int bedroomCardinality = cardinality("Bedroom");

  relation rooms : Room[0..5];
}

type Room;

type Bedroom : Room;
type Bathroom : Room;
```

Parent

The `parent` proxy variable refers to any attribute in the parent or ancestor variable. The first parameter identifies the attribute name in the parent. The second, optional parameter specifies the level. The type can reach up multiple levels, beyond the immediate parent.

Use this format:

```
parent(<parent variable name>, <level>)
```

In this example, the `parent` proxy variable specifies that the type `Room` and type `Bathroom` inherit the color of type `House`.

```
type House {
   string color = ["Red", "Green", "Blue"];

   relation rooms : Room[0..5];
}

type Room {
   string color = parent(color); // room's color inherits the
//color of the house

   relation bathrooms : Bathroom[0..1];
}

type Bathroom {
   string color = parent(color, 1); // bathroom's color inherits
the color of the house
}
```

this.quantity

The `this.quantity` proxy variable refers to the quantity of the current instance. The `this.quantity` proxy variable can be used only in the calculation rule. This variable differs from the `cardinality` proxy variable, which includes the quantity of other instances of the same type.

In this example, the `this.quantity` proxy variable specifies the quantity of rooms.

```
type Room {
   int qty = this.quantity;
   decimal(2) price;
   decimal(2) totalPrice = qty * price;
}
```

Variable Annotations

You can annotate variables with properties.

In this example, the `color` variable is annotated to indicate that it's configurable and has a default value of `"Red"`.

```
@(configurable = true, defaultValue = "Red")
string color = ["Red", "Green", "Blue"];
```

Variable Annotations

Property	Values	Description
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 devaultValue is not specified, the attribute picklist 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 do not update.
sequence	integer	Indicates the sequence in which the variables are configured. The constraint engine assigns the values based on the sequence, with the lowest in the sequence assigned first. For 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"];`
		`}`

sourceAttribute	Variable name in string	Sets the domain of the current variable to be the domain of the source variable.

tagName	string value	Value can be a comma-delimited string. Maps the tag from context definitions to the type attribute in the model. For 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");`
		`}`

Relationships

Relationships define how different types relate to each other.

In this example, type `House` is related to multiple type `Room` entities, with a specified quantity range of `[1..5]`, meaning that a house must have at least one and at most five rooms. In

addition, the cardinality (number) of type `Bathroom` is `2`, meaning that a house must have exactly two bathrooms (both the minimum and maximum number of bathrooms is two).

The `order` keyword specifies the order of child types for the relationship. The engine uses the order to instantiate the instance of the type. For example, if you specify one instance of `Room` in `rooms`, the engine creates `LivingRoom`. If you specify two instances of `Room`, the engine creates one `LivingRoom` and one `Bedroom`.

```
type House {
   relation rooms : Room[1..5] order (LivingRoom, Bedroom);
   relation bathrooms : Bathroom[2];
}

type Room;
type LivingRoom : Room;
type Bedroom : Room;
type Bathroom : Room;
```

The `order` keyword is optional. This example of the `rooms` relationship doesn't include `order`:

```
type House {
   relation rooms : Room[1..5] (LivingRoom, Bedroom);
}

type Room;
type LivingRoom : Room;
type Bedroom : Room;
```

Relationship Annotations

You can annotate relationships, as in this example.

```
@(configurable = true)
relation components : Component;
```

Relationship Annotations

Annotation	Values	Description
closeRelation	true, false	If the value is true, prevents the addition of new line items to the relationship. false is the default value.
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.
sequence	integer	Indicates the sequence in which the relationship is configured and executed.
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.

Supported Constraints

These constraints are supported. For more information, see the linked topic for each constraint.
Logical Constraints
Implication Operator (→)
Table Constraint
Message Rule
Preference Rule
Require Rule
Exclude Rule
Hide/Disable Rule
Action Rule

How User Input Order Affects Constraint Engine Behavior

The order in which a user sets attribute values for a product in PCM affects how the constraint engine computes the results.

For example, in this constraint for a laptop, Display and Display_Size are attributes. The constraint specifies that when Display is 2K, Display_Size should be 24 Inch.

```
 constraint(Display == "2K Built-in Display"  -> Display_Size == "24 Inch", "2K -> 24 Inch")
```

When a user adds a laptop to a quote and configures it, the constraint engine delivers different results depending on the order in which the user sets values for the attributes.

- If the user sets Display to 2K, the constraint engine updates the Display_Size to 24 Inch, to validate the constraint.
- If the user sets Display_Size to 15 Inch, then sets Display to 2K, the Display_Size of 15 Inch violates the constraint. The constraint engine never overrides or modifies a user-selected value. Instead of updating the Display_Size to 24 Inch, the constraint engine displays an error. To resolve the error and validate the constraint, the user can update Display to a value that is not 2K, or update Display_Size to 24 Inch.

Note: For the exclude rule, the constraint engine overrides user input when necessary to validate the constraint requirements. See Exclude Rule.

Logical Constraint

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:

- and (&&)
- conditional operator (?)
- implication (→)
- logical equivalent (<->)
- or (||)
- xor (^)

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.

In this example, the constraint uses the conditional operator to set the `windowShape` based on an equivalence comparison of `color`.

```
define COLORS ["Red", "Blue", "White", “Green”]
define MAX_ROOM 10
define MAX_ROOM_SIZE 100
type Room {
   decimal(2) width = [1..MAX_ROOM_SIZE];
   decimal(2) length = [1..MAX_ROOM_SIZE];
   decimal(2) area = width * length;
   string color = COLORS;
   string windowShape = [“Oval”, “Square”];
   constraint(color == “Red” ? windowShape == “Oval” :
windowShape == “Square”, “If color is red, windows should be
oval. Otherwise they should be square.”);
```

This example shows how to alter an attribute to different values in the if and else portions of the conditional operator.

```
   constraint(color != “Blue” && color != “Green” ? width <= 40
: width > 40);
```

The logical constraint has this syntax.

```
constraint(logic expression, string literal | string variable,
arg, …, arg);
constraint(logic expression, string literal | string variable);
constraint(logic expression);
```

Basic logical expressions are math expressions with variables and math operators, such as one of these:

- addition (+)
- subtraction(-)
- multiplication (*)
- division(/)
- remainder (%)

Basic logical expressions can use relational operators, such as one of these:

- equal (==)
- not equal (!=)
- greater than (>)
- greater than or equal to (>=)

- less than (<)
- less than or equal to (<=)

A logical constraint can cast variables from one data type to another. Each constraint takes an optional string variable or string literal, as the failure explanation if the constraint is violated. The failure explanation could be string format with additional arguments. CML supports two string formats. One is the java string format and another is a string with {} placeholder. The constraint solver uses the Java string format method to format the string or replace the placeholder with the actual argument value.

In this example, the first constraint specifies that, if there are more than two rooms, one of them must have the value `"Red"` for `color`. The {} placeholder is used for the failure explanation The second constraint ensures that the value for `size` must be greater than 2000.

```
type House {
   relation rooms : Room[2..10];
   expectedColor = "Red";

   constraint(rooms[Room] > 2 -> rooms.count(color == "Red") >
0, "one of the rooms must have the {} color)", expectedColor);
   constraint(rooms.sum(size) > 2000);
}

type Room {
   string color = ["Red", "Green", "Blue"];
   decimal width;
   decimal length;
   decimal size = width * length;
}
```

Implication Operator (→)

The implication operator `->` enforces a conditional constraint such that, if a precondition is satisfied, then the postcondition must also be satisfied, but if the precondition is not satisfied, the postcondition doesn't need to be satisfied. This operator is equivalent to the statement, "If a precondition holds, then a postcondition must also hold."

In this example, the implication operator `->` specifies that, if the value of `color` is `"Red"`, then the value of size must be `"S"`. The equality operator `==` specifies the variable values for the precondition and postcondition.

```
type House {
   string color = ["Red", "Blue", "Green"];
   string size = ["XS", "S", "M", "L"];

   constraint((color == "Red") -> (size == "S"));
}
```

Table Constraint

The table constraint defines valid combinations of values, specified in rows.

The table constraint has this syntax.

```
table(variable, …, variable, {value, .. value}, …, {value, …, value});
```

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

In this example, the table constraint specifies valid values for each of the house dimension variables, in separate rows.

```
type House {
   int width = [1..9];
   decimal(2) height;
   decimal(2) depth;
   string size = ["XXS", "XS", "S", "M", "L", "XL", "XXL"];

   // table constraint
   constraint(
       table(width, height, depth, size, // table columns
           {1, 1.25, 1.25, "XXS"},       // table rows
           {2, 1.5,  2.5,  "XS" },
           {3, 2.0,  3.0,  "S"  },
           {4, 6.0,  6.0,  "M"  },
           {5, 7.0,  8.0,  "L"  },
           {7, 9.0,  13.0, "XL" },
           {9, 10.0, 18.0, "XXL"}
       ));
  }
```

Message Rule

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

Message rules have the following 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`. At runtime, each message severity type behaves as follows:

- 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. For example:

```
constraint(laptop[Laptop].Display_Size == "27 Inch", "display {}
be 27 inches. This is a {} message", "must", "failure");
```

In this example, if the number of `rooms` is greater than `8`, a message is displayed with the number of rooms, as information only.

```
type House {
   relation rooms : Room[0..10];
```

```
   message(rooms.size > 8, "That's %d rooms!", rooms[Room],
"Info");
}
```

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 include at least five red rooms in the house, and delivers a message if they don't meet that preference.

```
type House {
  relation rooms : Room[0..10];

  // with explanation message
  preference(rooms.count(color == "Red") > 5, "Failure: There
are fewer than five red rooms");
}

type Room {
  string color = ["Red", "Blue", "Green"];
}
```

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 of Info severity, for 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);
```

In this example, the require rule specifies that if the house has more than five rooms, it must include a media room of designated dimensions and color, and delivers a failure message if a specific media room can't be included.

```
type House {
   relation rooms : Room[0..10];

   require(rooms.size > 5, rooms[MediaRoom]{ width = 5, height = 7, color = "Red" });

   // with explanation message
   require(rooms.size > 10, rooms[MediaRoom], "Failure: Big houses should have a media room");
}

type Room;
type MediaRoom : Room;
```

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]);
```

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.

Note: The exclude rule and the exclusion constraint both use the `exclude` keyword, but perform different functions. The exclusion constraint excludes a value from a relationship variable.

In this example, the exclude rule automatically removes the `MediaRoom` from the type `House` if there are fewer than five rooms.

```
type House {
   relation rooms : Room[0..10];

   exclude(rooms[Room] < 5, rooms[MediaRoom], "Can't have a
media room if there are fewer than 5 rooms in total");
}

type Room;
type MediaRoom : Room;
```

Hide/Disable Rule

Hide or disable a component in a bundle, an attribute, or an attribute value when certain conditions are true, to remove the element from view, or to disable selections for it.

- 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)
```

Variables for Hide/Disable Rule

Variable in Rule	Usage	Example
declaration	Condition upon which the constraint occurs	`Display_Size==”24 Inch”`
action	Designates whether rule is hide or disable	`“disable”, “hide”`
actionScope	Designates whether the rule acts on an attribute or relationship scope	`“attribute”, “relation”`
actionTarget	Designates the specific variable that the rule acts on	`“storage”, “software”`
actionClassification	Designates whether the rule acts on a type or a value	`“type”, “value”`
actionValueTarget	Designates the type or value that the rule acts on	`“Quip”, [“Cloud Storage Enterprise - 2 TB”, “SSD Hard Drive 1TB”], “SSD Hard Drive 1TB”`

A hide or disable 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.

These examples use the hide rule. To use similar code for disable rules, replace `“hide”` with `“disable”`.

Hide a component:

```
relation productivitySoftware : Software;
```

```
// Hide the quip component as an acceptable software choice
rule(mouse.Wireless == true, "hide", "relation",
"productivitySoftware", "type", "Quip");
```

Hide an attribute:

```
// Hide Attribute
rule(Display_Size == "24 Inch", "hide", "attribute", "Memory");
```

Hide an attribute value:

```
// Hide Attribute Value
rule(Display_Size == "24 Inch", "hide", "attribute", "Storage",
"value", "Cloud Storage Enterprise - 2 TB");
```

Action Rule

The action rule is used to define simple conditions and actions. When the condition is met, the constraint solver engine raises the action with parameters and the caller can execute the action with the parameters. The action can be anything the caller defines.

Action rules have the following 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 or any custom code. 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.

This example uses the Product Configurator API:

```
type accessory;

type Case : accessory;

type Charger : accessory;

type Wallet : accessory;

type WirelessPlan {
   @(defaultValue=true)
   boolean highEnd;
   @(defaultValue="iPhone")
   string phone = ["iPhone", "Xiaomi", "Galaxy"];
   int price;

   relation accessories : accessory {
   }

   constraint(table(phone, price, {"Xiaomi", 900}, {"iPhone",
1000}, {"Galaxy", 1100}));

   // HIDE ACTION RULE

   // [Attribute] Hide price attribute
   rule(highEnd == true, "hide", "attribute", "price");

   // [Attribute Value] Hide "Xiaomi" attribute value from phone
   rule(highEnd == true, "hide", "attribute", "phone", "value",
"Xiaomi");
   rule(highEnd, "hide", "attribute", "phone", "value",
["Xiaomi", "Galaxy"]);

   // use java String.format to format the string
   message(highEnd, "High end phone %s", phone, "Warning");
   // use {} as parameter placeholder
```

```
   message(highEnd, "High end phone {}", phone, "Info");

   // [Product] Hide Wallet from accessories
   rule(phone != "iPhone", "hide", "relation", "accessories",
"type", "Wallet");

   //
-------------------------------------------------------------
   // DISABLE ACTION RULE
   // [Attribute] disable the price attribute
   rule(highEnd == true, "disable", "attribute", "price");

   // [Attribute Value] disable the "Xiaomi" attribute from
phone attribute - If phone is a Picklist attribute, the “Xiaomi”
option is just removed from the dropdown in the UI
   rule(highEnd, "disable", "attribute", "phone", "value",
"Xiaomi");

   // [Product] disables Wallet from accessories
   rule(phone != "iPhone", "disable", "relation", "accessories",
"type","Wallet");

   //
-------------------------------------------------------------
   // MESSAGE ACTION RULE (INFO, WARN, ERROR)
   message(phone != "iPhone", "Your phone isn't an iPhone.",
"info");

   message(phone != "iPhone", "Your phone isn't an iPhone. Your
phone: {}", phone, "warning");

   message(phone != "iPhone", "Your phone isn't an iPhone. Your
phone: %s", phone, "error");
}
```

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, wich 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 conrstraint 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.

7: Automatically Adding 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”);
```

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.