Tutorial #5 - Automated Configuration

Introduction

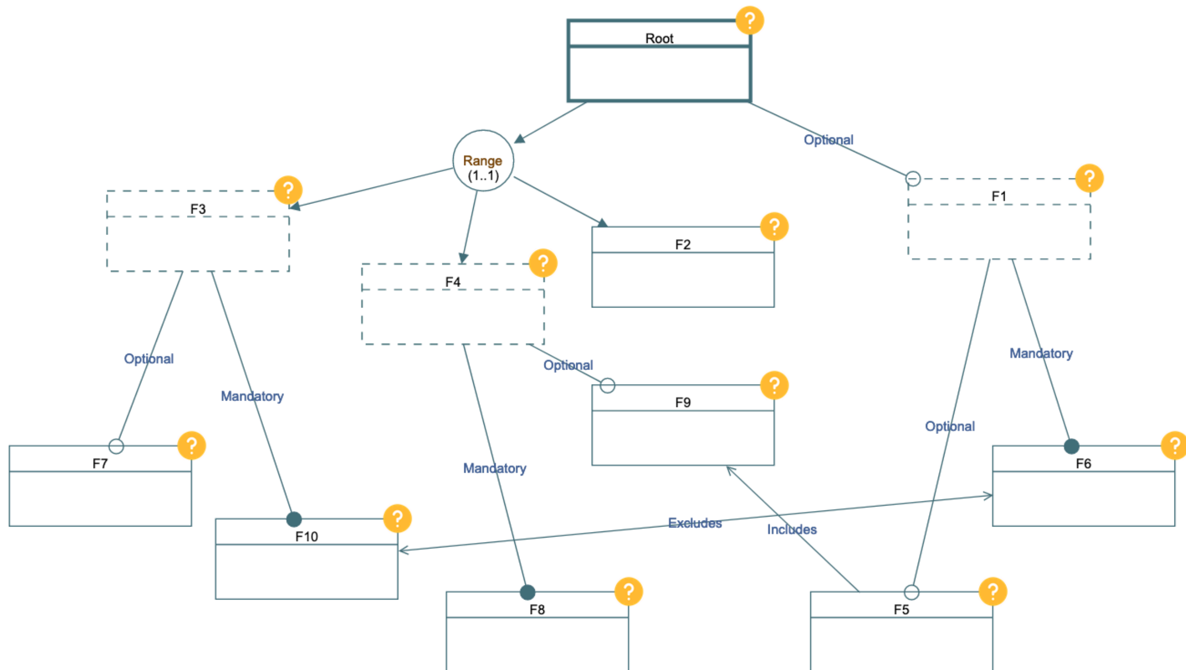
This tutorial aims to demonstrate the mechanisms for automated configuration of Variability Models through the use of the VariaMos PLEIADES back-end. This back-end service allows you to perform configuration tasks on your models using a JSON-based query specification system.

Prerequisites:

1. An internet connection and access to VariaMos at: https://develop.variamos.com/
2. Your instructor will notify you if the server is available, if not or it is inaccessible from your network, you will need to run the translation tool locally. The currently available endpoint is:
   http://ec2-3-130-187-131.us-east-2.compute.amazonaws.com:5000/query
3. If the endpoint is inaccessible, you must run the server locally. To do so you must have **docker** installed. Here's a short tutorial to set up docker (in french): https://docs.google.com/document/d/1LsH29Ku7TtSj9r3b5oTuGAeFEjA1eyxNPJ-iho8jGX4/edit?usp=sharing
4. (With docker) Simply run:
   docker run -it -p 5000:5000 ccorre20/semantic_translator
5. (Without docker): Ask your instructor for the API endpoint to use in the following sections. By default the endpoint is http://ec2-3-130-187-131.us-east-2.compute.amazonaws.com:5000/query as mentioned above.

Part 1: Configuration Tasks

Consider the following model:



You can download it from the following link:
https://drive.google.com/file/d/1wbOBdHW0LrZPJKrx07kBg0LboXdHMQWn/view?usp=sharing

Load it into Variamos. Once you've done this, we can now begin exploring the configuration capabilities offered to us by the system. As before with the analyses we performed over the different models in Tutorial 4, our primary interest in automation is tied to the need to be able to reason over large models with the least amount of manual effort. In the case of configurations, we can identify 3 general tasks that are of interest:

1. Identify one or more specific configurations if they exist (and visualize them), or be notified of their absence otherwise.
2. Complete (and visualize) a partially defined configuration or notify its unsolvability (lack of satisfying solution) given the constraints in the model.
3. Calculate and order configurations according to some optimization function.

Tasks 1 and 2 share a particularly important aspect in common: the same queries are used for both, the only difference is whether or not specific elements are marked as selected (or deselected). Recalling what was shown in Tutorial #4, the query for a single solution is as follows:

```
{
 "solver": "swi",
 "operation": "solve"
}
```

The query for finding ***n*** solutions, with n=2 in this example is as follows:

```
{
 "solver": "swi",
 "operation": "nsolve",
 "operation_n": 2
}
```

Let's now consider the effect of running our very first query on the above model by running it as follows:

Queries

Query | Results | CLIF Semantics | Solver Specific Semantics | Saved Queries

Translator Endpoint

http://develop.variamos.com:5000/query

Enter the adress of the endpoint to use for the queries.

Query

{"operation":"solve","solver":"swi"}

Enter Query Name

Save Query

Close | Submit Query | Sync CLIF Semantics | Reset model configuration state

We should then observe the following in the results tab:

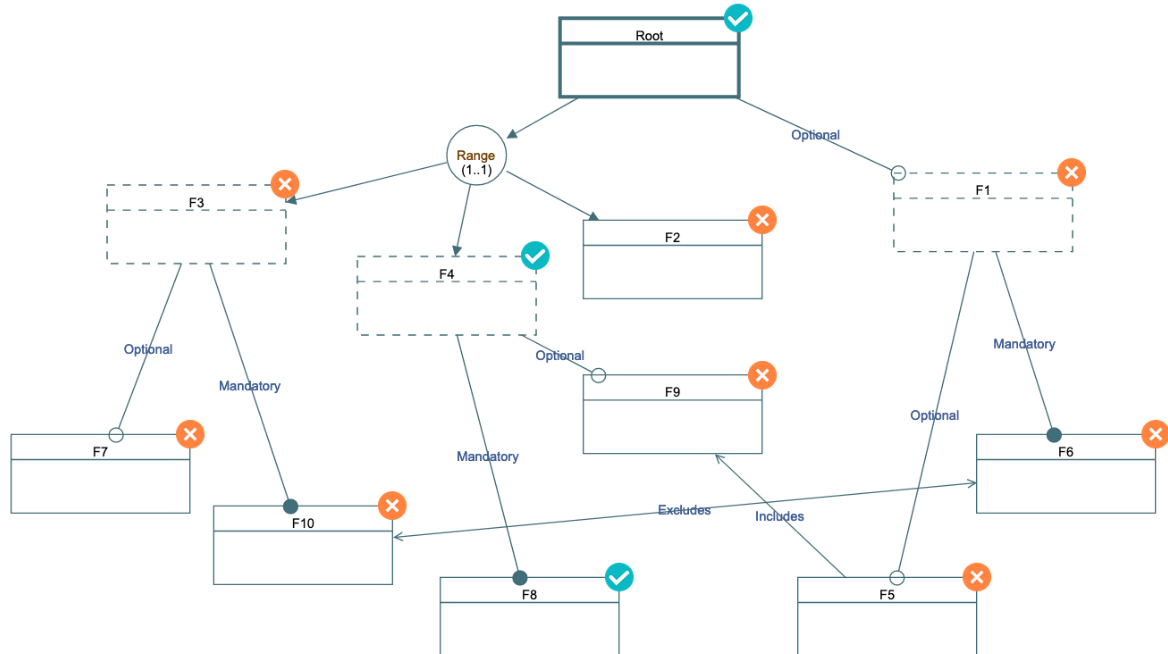
Queries

Query | Results | CLIF Semantics | Solver Specific Semantics | Saved Queries

Solution 0 | Visualize

Close | Clear Query Results | Reset model configuration state

Do note that "Solution 0" does not mean that 0 solutions were found: it merely means that this is the response to the very first query we sent (counting from 0 as is customary in programming languages). If we click on "Visualize", we'll observe a specific configuration applied to the model:



Now that we've obtained this configuration, if we inspect it, we can indeed conclude that it satisfies all the constraints in the model. Now, we might well like to see other possible configurations as determined by the solver, which we'll examine next.

N.B.: Given the nature of the solving procedure, it is not simple (and occasionally outright impossible) to control the order in which solutions are calculated, since, as a model and its space of possible configurations grows, the solver can only, in practice, traverse a limited portion of it in a reasonable amount of time. This is not a technical limitation, but rather a fundamental limitation with any automated reasoning task whose underlying nature is inherently combinatorial.

Let's now return to the query modal and click on "Reset model configuration state" to return the model to its default state. Let's now run the second query outlined above, and ask for more (in this case 2) configurations. To do this load the second query into the "Query" box as depicted below:

Queries

Query | Results | CLIF Semantics | Solver Specific Semantics | Saved Queries

Translator Endpoint

http://develop.variamos.com:5000/query

Enter the adress of the endpoint to use for the queries.

Query

{"solver":"swi","operation":"nsolve","operation_n":2}

Enter Query Name

Save Query

Close | Submit Query | Sync CLIF Semantics | Reset model configuration state

If we run this query and return to the "Results" tab, we should observe the following:

Queries

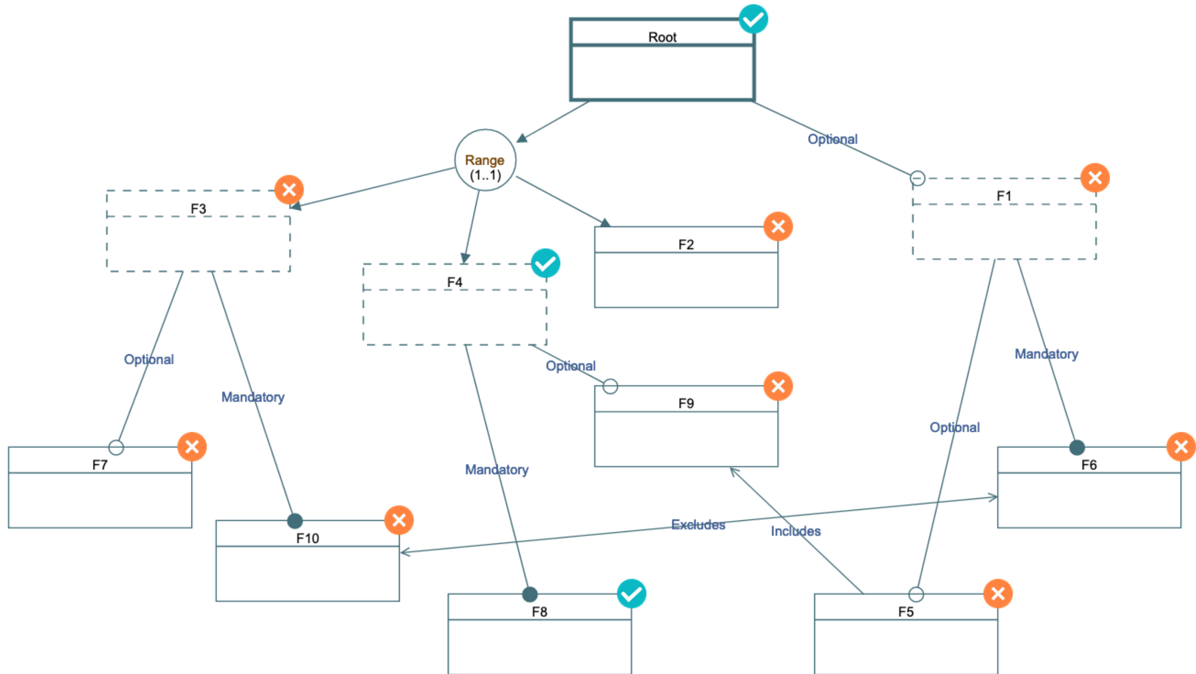
Query | Results | CLIF Semantics | Solver Specific Semantics | Saved Queries

Solution 0		Visualize
Solution 1	1 2	Visualize

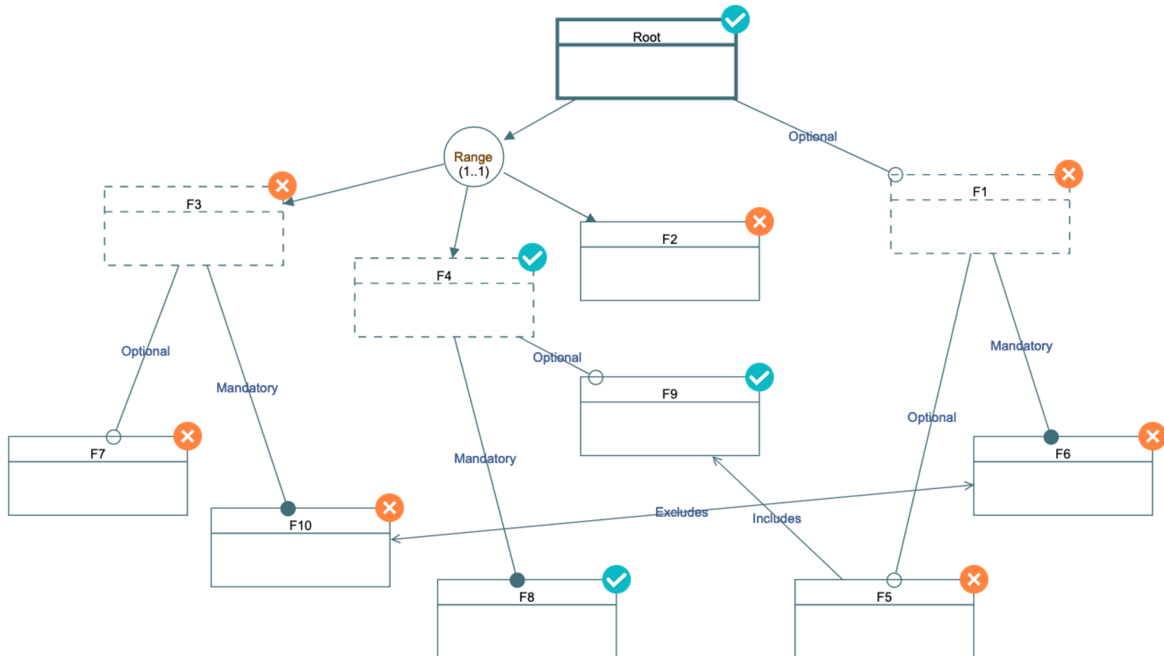
Close | Clear Query Results | Reset model configuration state

Which tells us that, unsurprisingly, there are two solutions, which we may visualize just like in part 3 of Tutorial 4. (You do not need to reset the model to switch between them, only to return the model to the default state.)

The two configurations we obtain are:



And

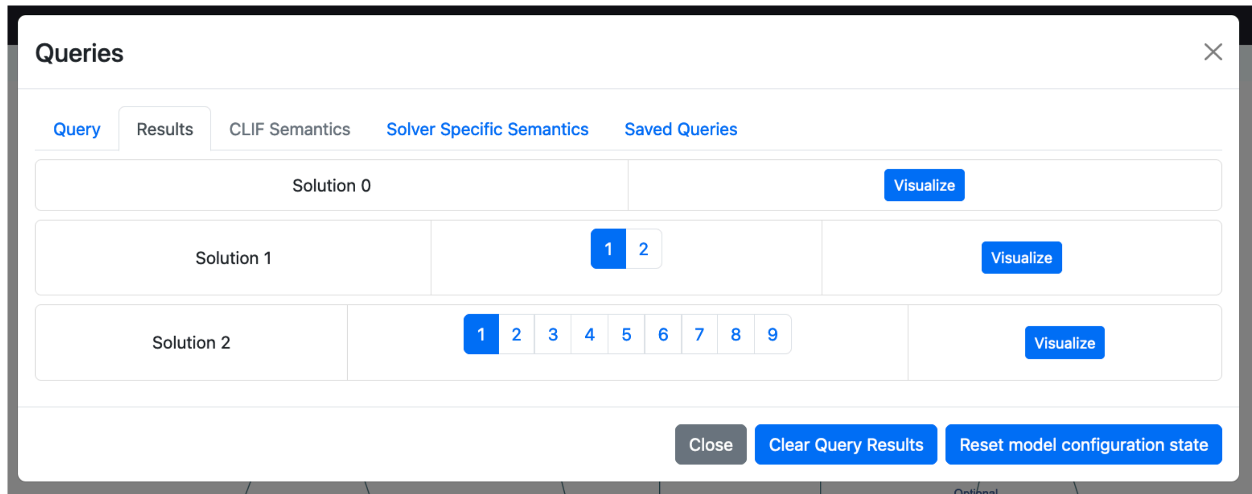


As can be seen, the second of these configurations is a small variation of the first with the inclusion of F9.

If we ask for, for example, 10 configurations, by changing the query to the following:

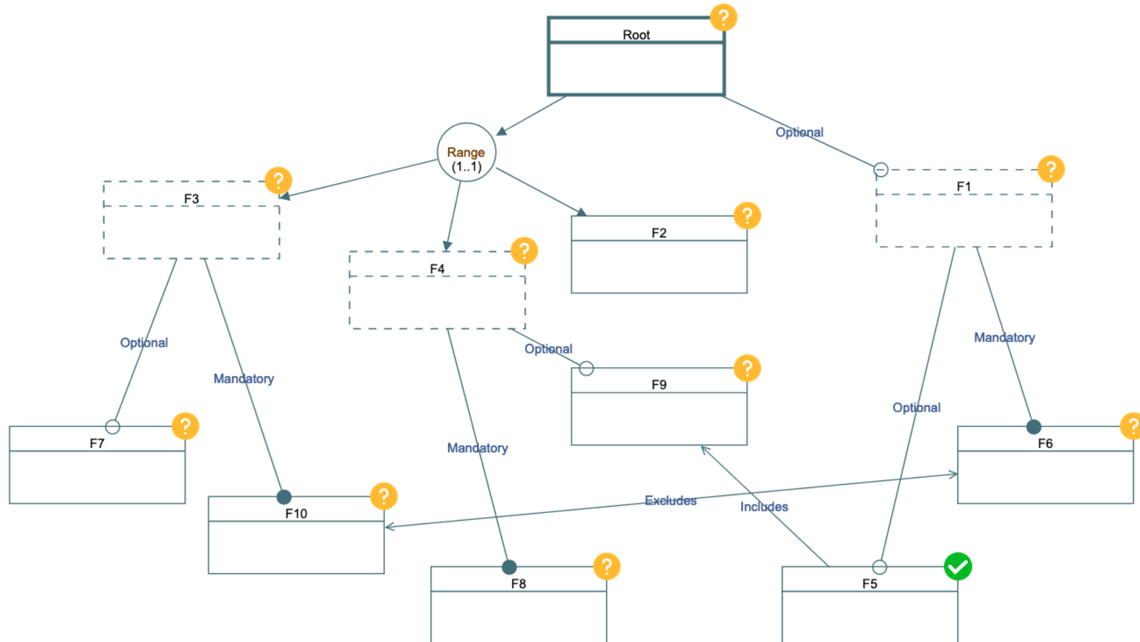
```
{
```

```
 "solver": "swi",
 "operation": "nsolve",
 "operation_n": 10
}
```
We will observe the following result:

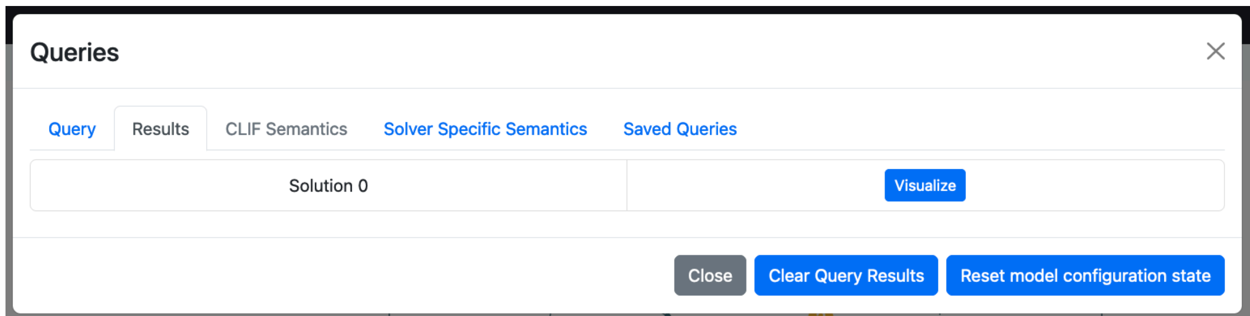


Which, recalling what was said in Tutorial #3, tells us that there are 9 possible solutions to the model. This tells us, then, that, while we requested 10 solutions, only 9 really exist. We can, of course, visualize each of them just as we did with the previous query.

With this done, we can now turn to the second task mentioned above. Let's suppose we want to find a configuration that in addition to respecting all of our model's constraints, has a certain set of features selected. For the sake of our example, let's select the F5 feature (following the instructions provided in tutorial 4, that is, clicking on the question mark on the top right corner of F5 until we see the green check mark denoting "selected".

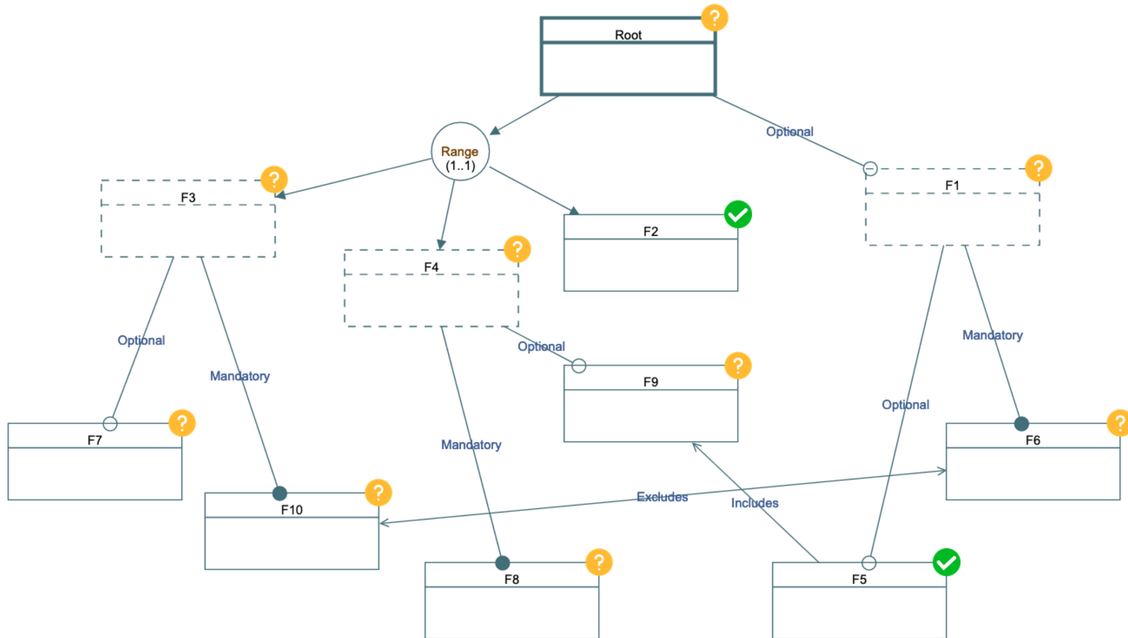


With this done, we can once more ask for 10 solutions as before and we observe the following (I previously cleared the query results, but this is optional):



What this indicates is perhaps a bit surprising: there is only one solution where F5 is selected, due to all of the other relationships present in the model.

If we continue with our experiment on this model, and set our configuration as follows, by selecting, for instance, F5 and F2 as shown:



And we once more rerun our query, we will observe the following:

Queries

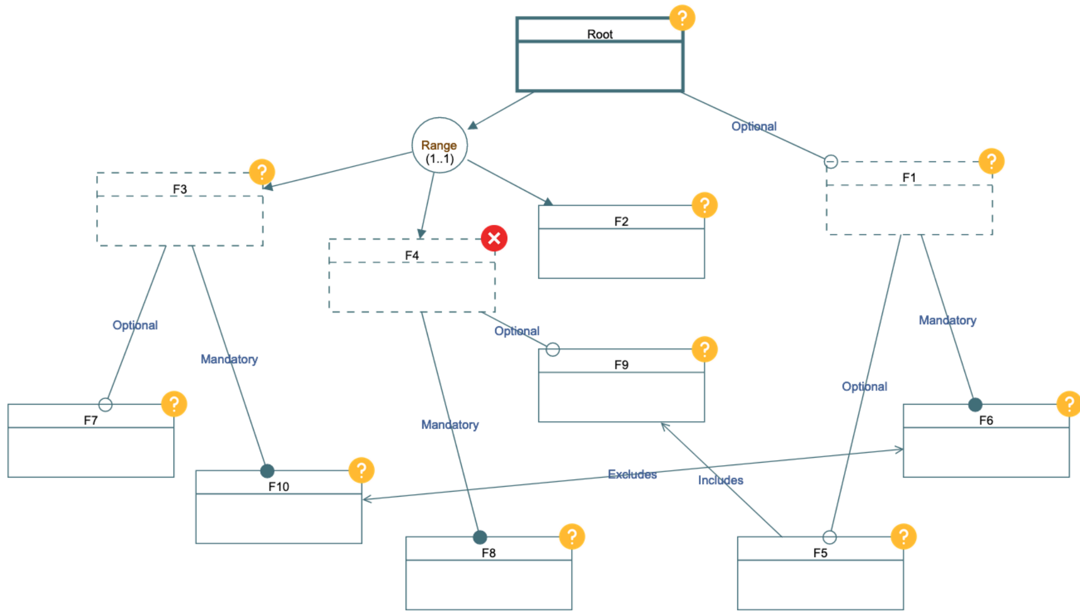
Query | Results | CLIF Semantics | Solver Specific Semantics | Saved Queries

Solution 0	Visualize
Solution 1	UNSAT

Close | Clear Query Results | Reset model configuration state

Which tells us that indeed F5 and F2 can never coexist, and, if, for example, this was a particular set of requirements we wished to have for a specific product, it would not be feasible and we must reconsider our model and its associated requirements.

We can rerun a similar experiment to that done with F5 by asking the opposite question, that is, deselecting a specific feature, for instance, asking for F4 **NOT** to be included as follows:



If we rerun our query from above for this partial configuration, we observe that there are only 4 products that satisfy this partial configuration.

Queries

Query | Results | CLIF Semantics | Solver Specific Semantics | Saved Queries

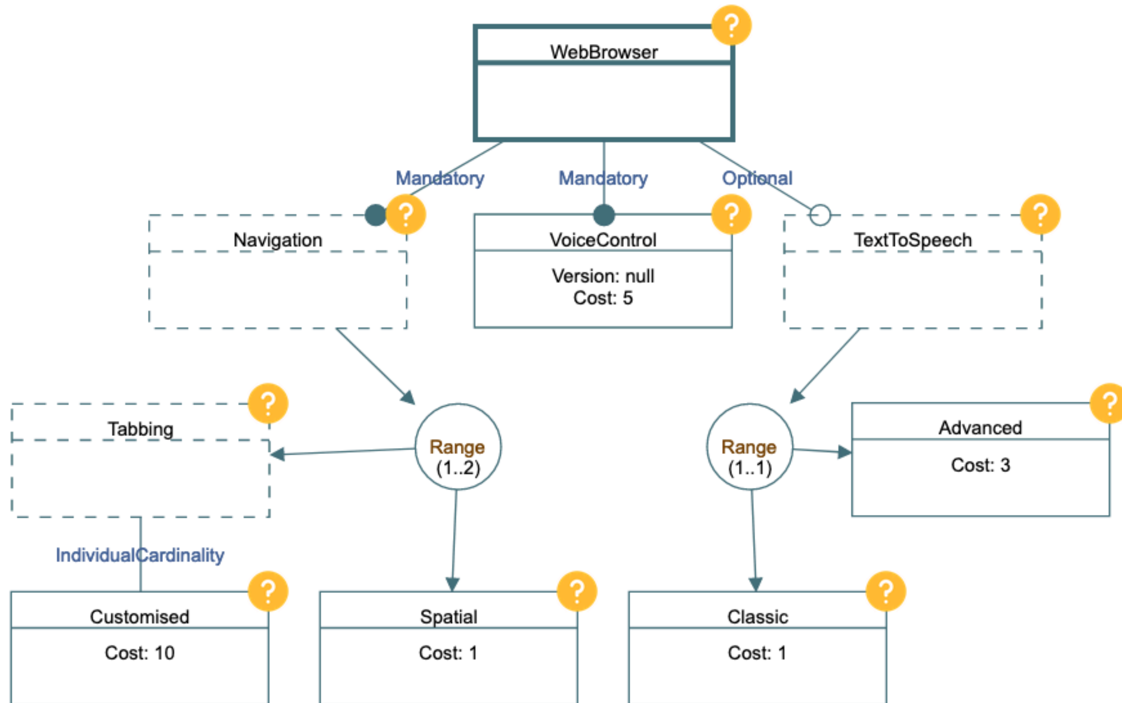
Solution 0		Visualize
Solution 1		UNSAT
Solution 2	1 2 3 4	Visualize

Close | Clear Query Results | Reset model configuration state

In any of these cases, it is of course possible to mix and match selected and excluded features in a given model, which will impact the set of possible products.

Part 2: More complex configuration tasks

To illustrate some more powerful capabilities of this system, and in particular cover our much more complex case of optimizing configurations according to different metrics, let's now augment the model used in Tutorial #4 with attributes representing their cost and assigning some values to each of the concrete features to obtain the following diagram:



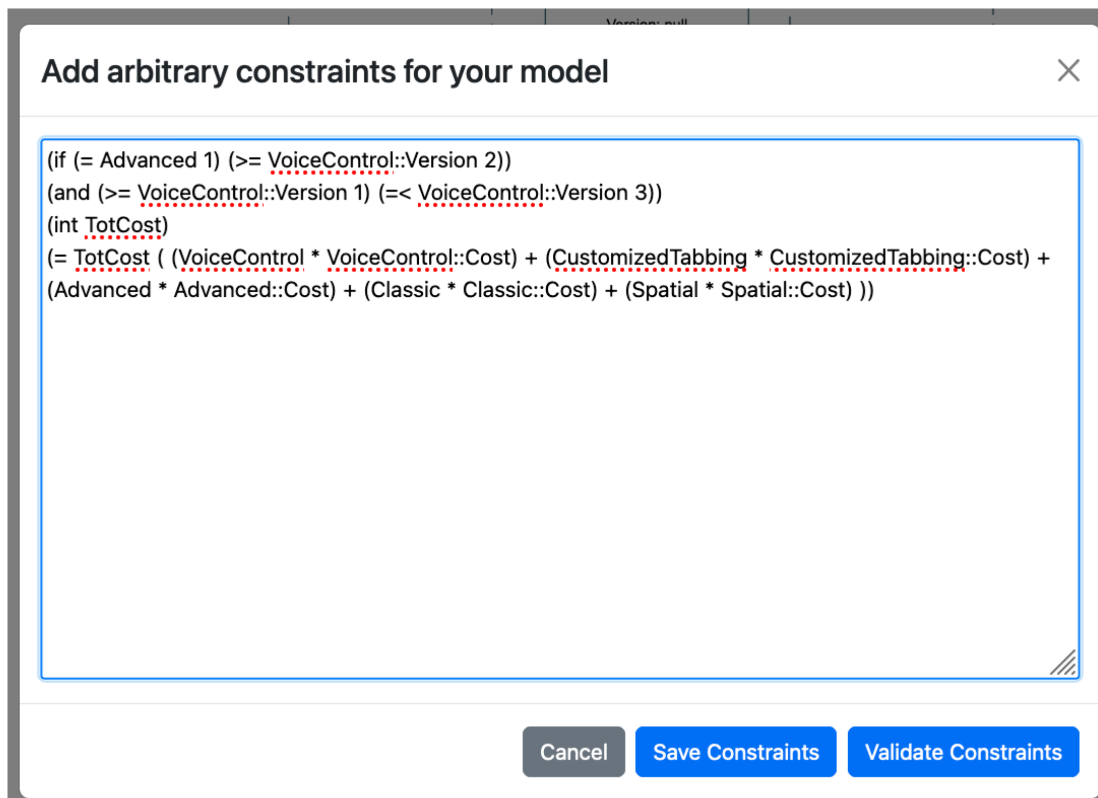
This modified model is available at (it is perhaps best to use this version to ensure that all works just as in the tutorial):
https://drive.google.com/file/d/1YVhY5HES18OXQnqAu2FYXJUE4YGDxqlI/view?usp=sharing
N.B. Due to a technical limitation, for this portion of the tutorial the "CustomizedTabbing" feature had to be renamed to "Customised" (if you want to use the modified model available on the above link, this change has already been done for you) to avoid a conflict with the "Tabbing" feature in one of the processing steps.

We can now use the optimization capabilities of our reasoning system (mentioned in Tutorial #4) to ***find out what is the most expensive product we can make within the product line*** based on these costs. To do so, we must first add a definition of what the notion of "total cost" is for our model. Intuitively it should be the cost of whatever

features we select. We can express this notion by adding the following constraints to the arbitrary constraints for our model:

```
(int TotCost)
(= TotCost  ( (VoiceControl * VoiceControl::Cost) + (Customised * Customised::Cost) +
(Advanced * Advanced::Cost) + (Classic * Classic::Cost) + (Spatial * Spatial::Cost) ))
```

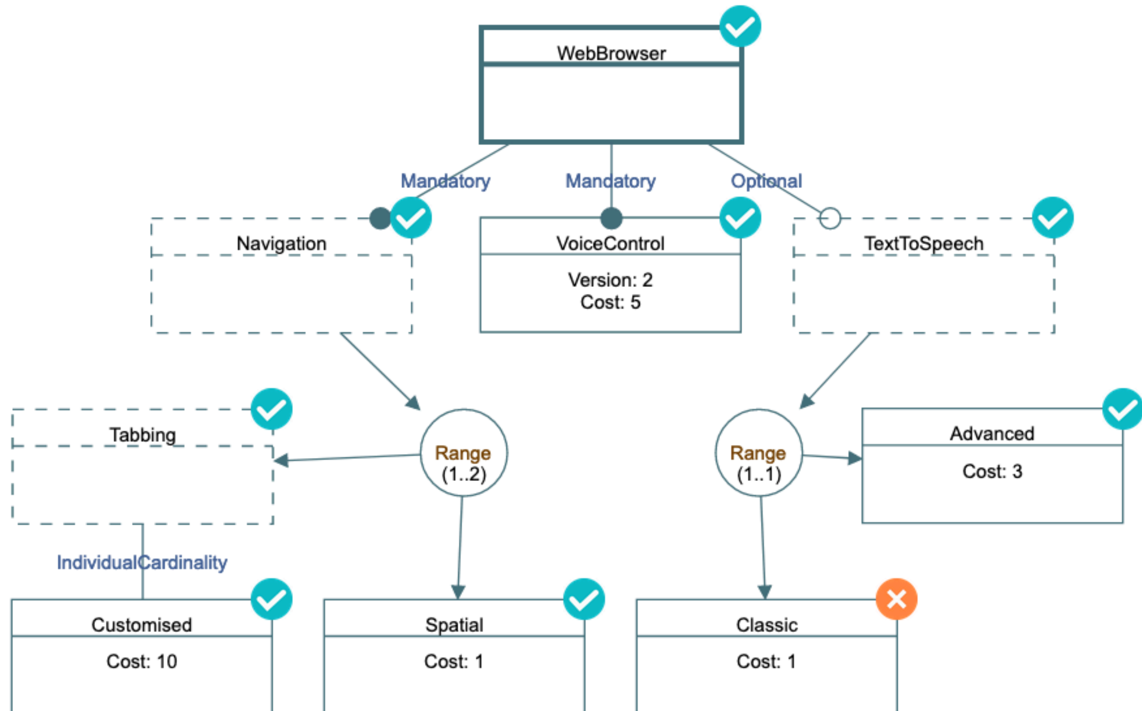
This encodes essentially the fact that the **TotCost** variable is an integer and is related to the sum of all the costs, each multiplied by the presence or absence of a given feature. We can then add this to the arbitrary constraints we defined for our model to obtain the following:



With these constraints we can now write an **optimization** query that will allow us to ***find the most expensive product*** (recalling our query definition explanation in tutorial #4):

```
{
   "operation": "optimize",
   "solver": "swi",
   "optimization_target": "TotCost",
   "optimization_direction": "max"
}
```

If we run this query and visualize the result we will obtain the following:

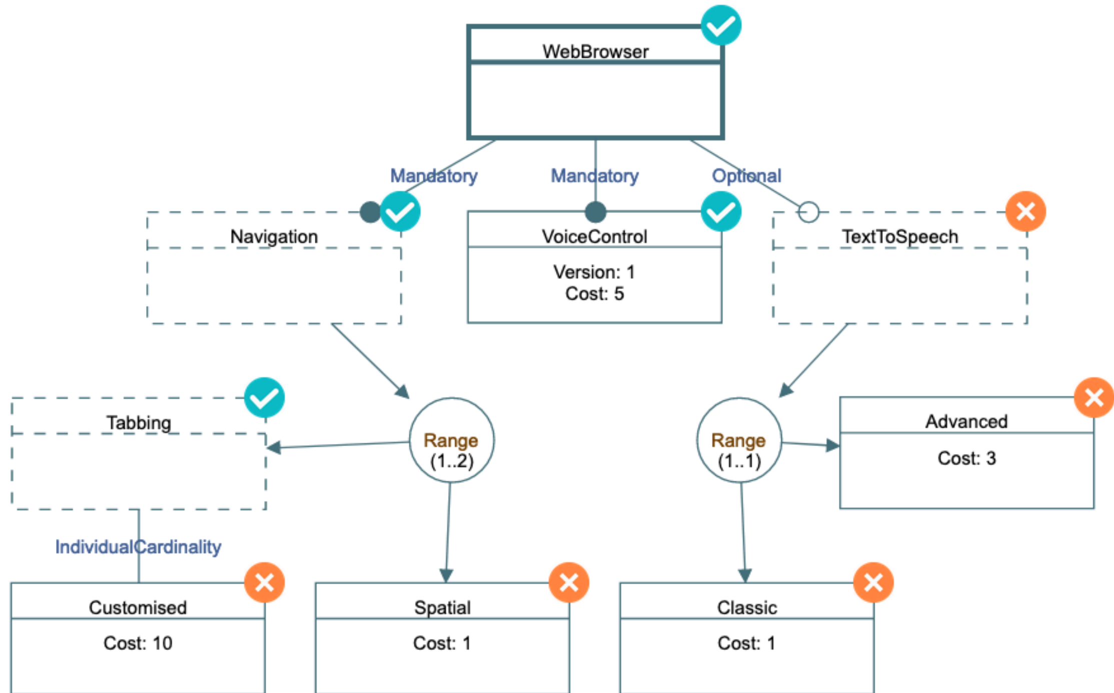


N.B.: As you can see, not only the features but also their attributes were automatically configured in such a way that it respects not only the variability requirements but also the requested optimal configuration requirement. However, the attributes resetting capability is not yet implemented in the tool, so, please reload your model before the next step and reinsert the arbitrary constraints.

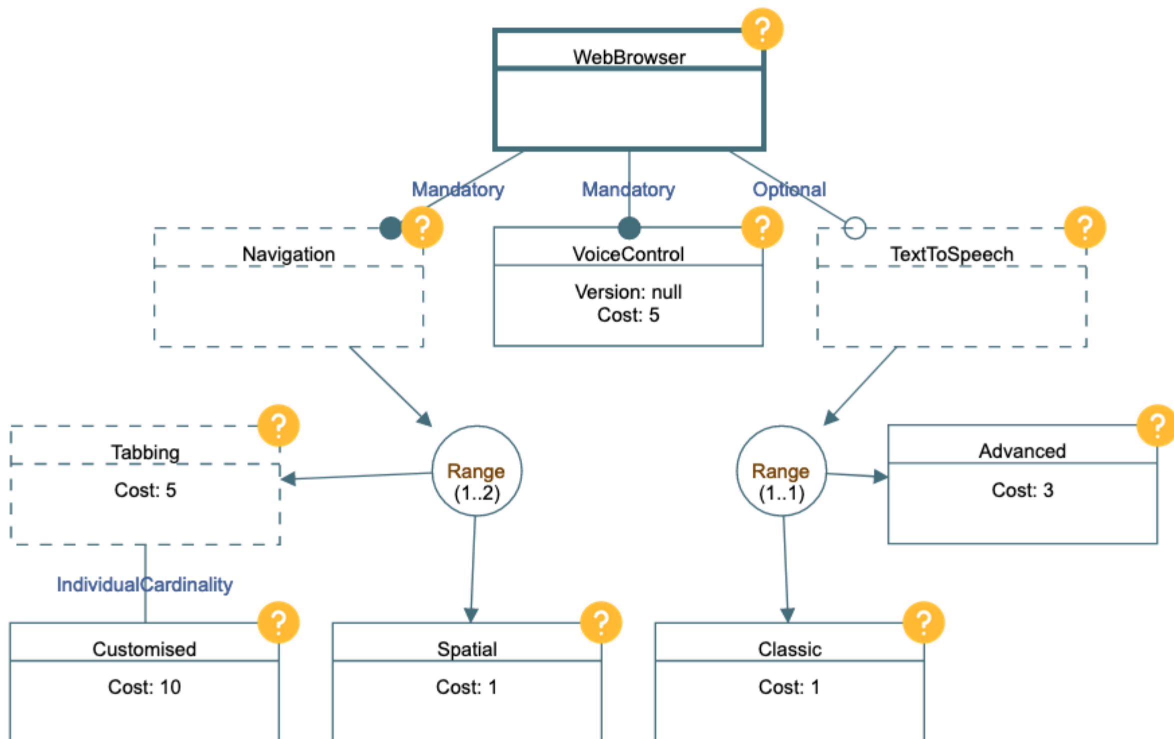
If we now ask the opposite question: **what is the cheapest product that can be produced by our product line? Using the following query:**

```
{
  "operation": "optimize",
  "solver": "swi",
  "optimization_target": "TotCost",
  "optimization_direction": "min"
}
```

If we run this query, we will obtain the following result:



An interesting point of note is that the cardinality we defined for "Customised" is defined as being from 0 to 3, because one can have either just normal tabbing as in most modern browsers, or up to three customised tabbing systems in addition to the basic one. Given this particular constraint, it should then be no wonder that this particular route is chosen. We can infer from this the following, our notion of cost isn't entirely well founded for the model, and that perhaps we need to associate a cost to our tabbing module to reflect this notion. Let's then **(REMEMBER TO RELOAD THE MODEL GIVEN THE ISSUE NOTED ABOVE)**, add a cost of 5 to the Tabbing feature as follows:



Now, let's modify our equation for total cost adding the cost of tabbing as follows:

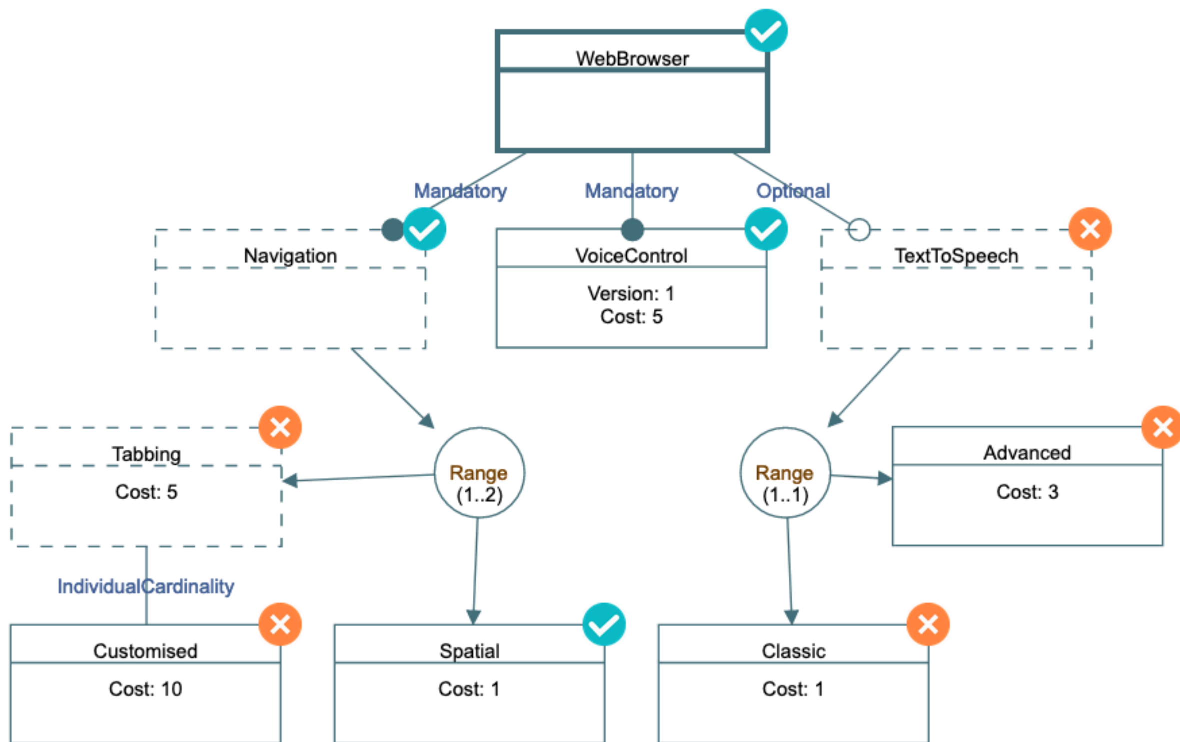
```
(int TotCost)
(= TotCost ( (Tabbing * Tabbing::Cost) + (VoiceControl * VoiceControl::Cost) +
(Customised * Customised::Cost) + (Advanced * Advanced::Cost) + (Classic *
Classic::Cost) + (Spatial * Spatial::Cost) ))
```

Let's add this again to our model constraints to obtain:

Add arbitrary constraints for your model

```
(if (= Advanced 1) (>= VoiceControl::Version 2))
(and (>= VoiceControl::Version 1) (=< VoiceControl::Version 3))
(int TotCost)
(= TotCost ( (Tabbing * Tabbing::Cost) + (VoiceControl * VoiceControl::Cost) + (Customised *
Customised::Cost) + (Advanced * Advanced::Cost) + (Classic * Classic::Cost) + (Spatial * Spatial::Cost)
))
```

And let's now once more run the query asking for the cheapest possible product to obtain the following solution:



As can be seen, "Tabbing" is no longer part of the cheapest possible product, since we now have a more detailed and accurate representation of the cost of the products. Nevertheless, if we analyze the first result we obtained, which corresponds to the most expensive product, this change has no effect and it remains so, as neither "Advanced" or "Classic", and have only effect on the "Navigation" portion of the diagram.

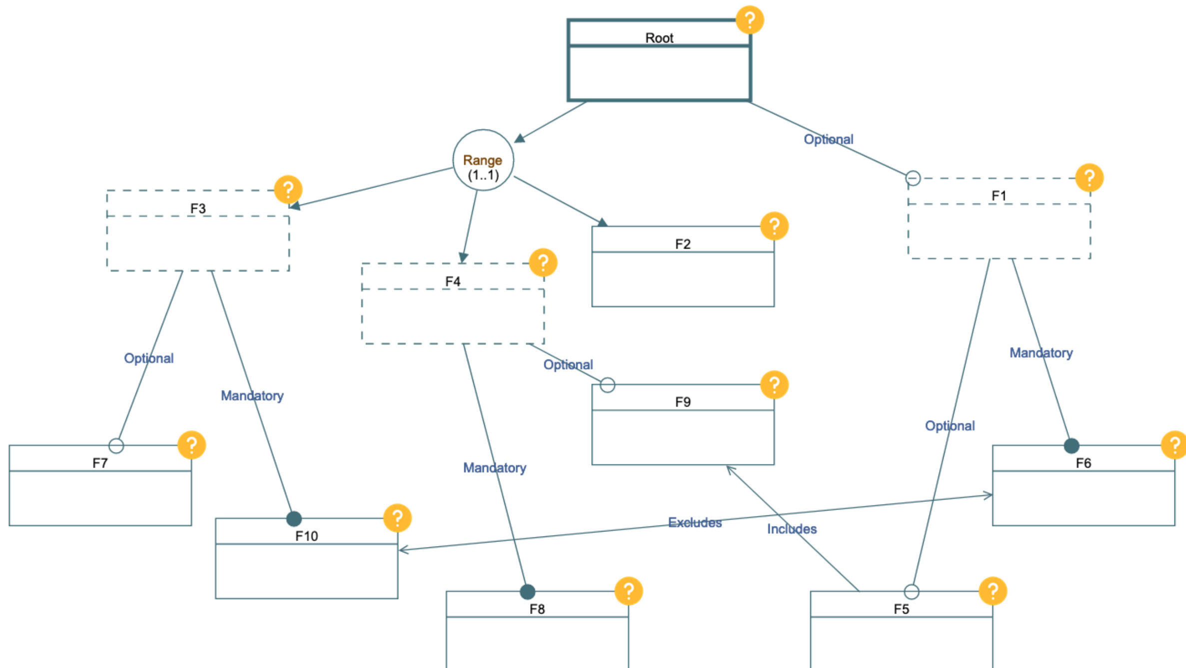
What this serves to illustrate is that even when working on configuration tasks, automated reasoning systems not only save us considerable amounts of time, but they also allow us to determine where we might have modeled incorrectly. However, it also serves to show that human involvement is also important, no matter the sophistication of the analysis tools, in order to interpret the results and to catch potential flaws in the problem posed to the analysis tools.

Part 3: Practical Exercise

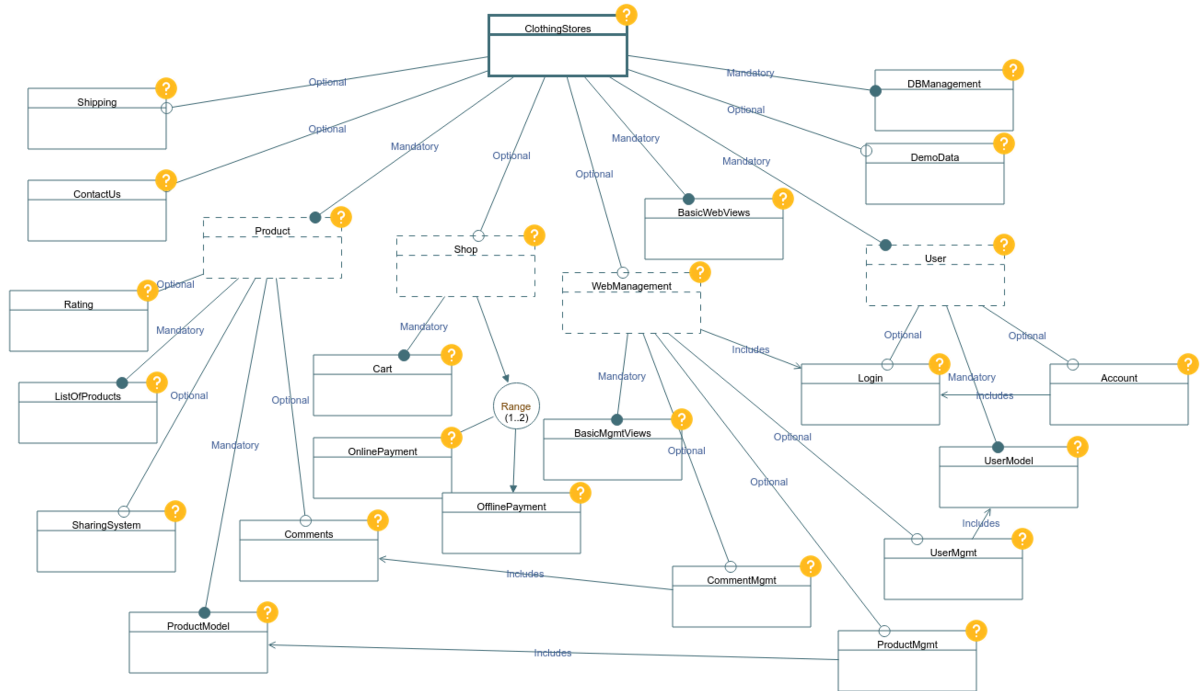
Please help us to improve the usability of this under-development tool answering the following usability test questionnaire: https://forms.gle/qu2vfyL3c8qTS8UY9

Please, do the following:

1. Reload the model used for Part 1 of this tutorial.



2. Calculate how many solutions there are when selecting "F1".
3. Calculate how many solutions there are when deselecting "F1".
4. Calculate how many solutions there are when selecting "F4" and deselecting both "F3" and "F1".
5. Reload the model used for Part 4 of Tutorial 4:



For your convenience, this model is provided to you in the following link: https://drive.google.com/file/d/1TVasQbclYRZZ_Rdzak47q1T9iyv4QJgh/view?usp=sharing

6. Insert the following constraint into your model. It defines the TotFeat variable as the total number of concrete features that are selected.

```
(int (0 10000) TotFeat)
(= TotFeat (Shipping + ContactUs + Rating + ListOfProds + SharingSystem +
ProdModel + ProductMgmt + Comments + Cart + OnlinePayment + OfflinePayment +
BasicMgmtViews + CommentMgmt + ProductMgmt + UsersHandling + UserMgmt + Login +
UserModel + BasicWebViews + Account + DemoData + DBManagement))
```

7. Using "swi", calculate the number of features in the product with the largest number of features.
8. Using "swi", calculate the number of features in the product with the smallest number of features. (Hint: change "max" to "min" in your query).