

Tutorial #3 - Code generation and solver-specific analysis

Introduction

This tutorial is intended to demonstrate the process of constructing and using the computational code of models built under the VariaMos modeling environment. A Variability Modeling Language's semantics are, put simply, the meaning one assigns to the elements of the language. In contrast, a concrete model's semantics are the set of satisfying assignments, that is, the valid products as determined by the model's elements and relations between them.

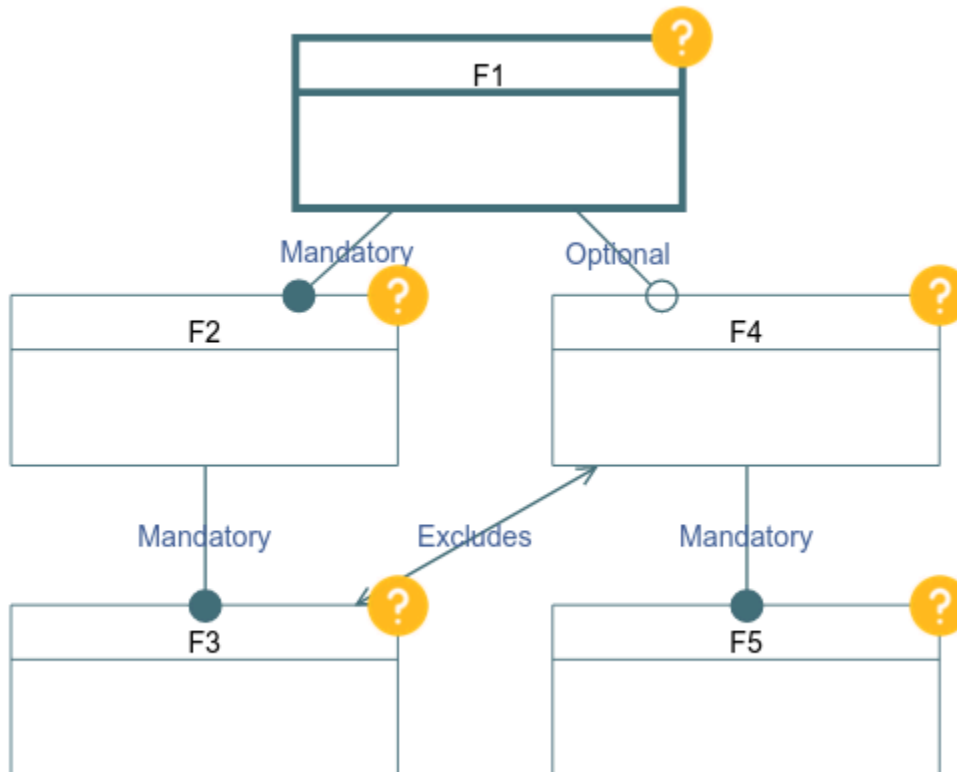
In this tutorial we will be working with very simple models in order to understand the basic mechanisms of creating models and obtaining their representations as programs.

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: Creating a simple feature model

For this portion of the tutorial we will be using the following simple model:



It is available for download in the following link:

https://drive.google.com/file/d/1mDFUfbjz7yUGM37Xi_7-OdFwamgb4g0G/view?usp=sharing

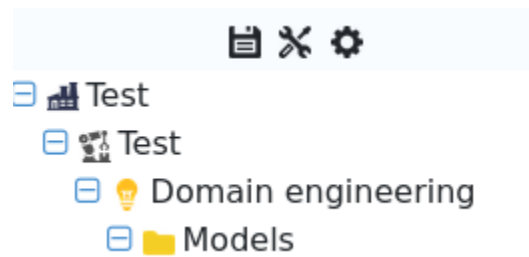
You can also create it yourself with the editor as explained in the previous tutorial.

Note: If you load the file and the names of the features are missing, simply click again on the model in the explorer and they should appear.

Part 2: Obtaining the model's formal representation

In order to get an executable representation of our model so we can perform analyses on it, we must obtain its concrete semantics. In this case we will be using the model's representation in SWI Prolog so that we can use its built-in constraint solver for analysis.

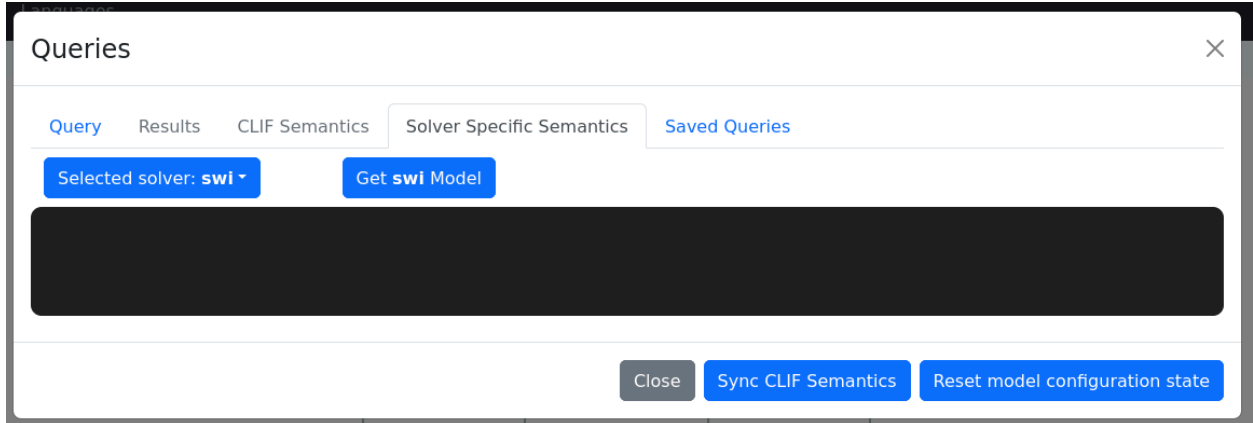
Begin by clicking on the “wrench and screwdriver” icon above the project explorer:



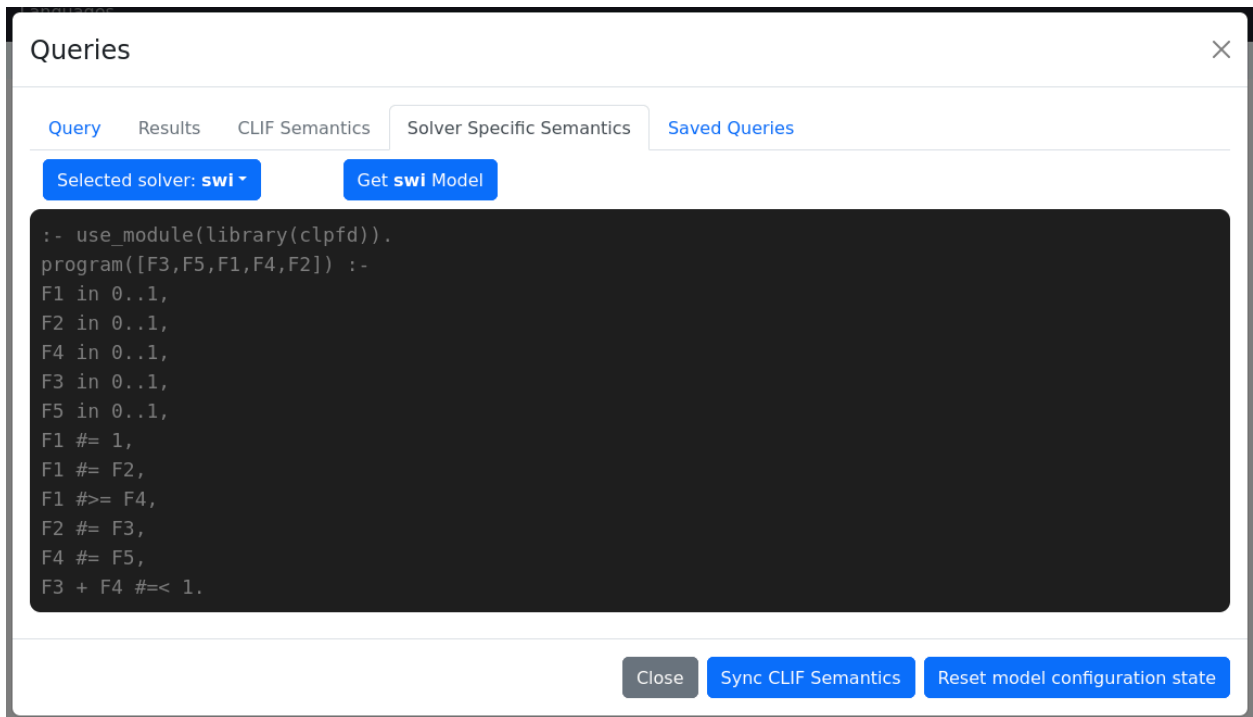
You should now see the following modal (possibly without the “Translator Endpoint” field, if you are using docker, the default endpoint should work):

A screenshot of a modal window titled 'Queries'. The window has a close button (X) in the top right corner. Below the title bar, there are five tabs: 'Query' (selected), 'Results', 'CLIF Semantics', 'Solver Specific Semantics', and 'Saved Queries'. The 'Query' tab is active and contains the following elements: a 'Translator Endpoint' label, a text input field containing 'http://localhost:5000/query', a small instruction 'Enter the address of the endpoint to use for the queries.', a 'Query' label, a large empty text area for the query, an 'Enter Query Name' label, a text input field for the name, and a blue 'Save Query' button. At the bottom of the modal, there are four buttons: 'Close' (grey), 'Submit Query' (blue), 'Sync CLIF Semantics' (blue), and 'Reset model configuration state' (blue).

Navigate to the “Solver Specific Semantics” Tab and select the language to which you want to transform your model. In this case, please select **swi** which corresponds to the SWI Prolog language and solver.



Now, click on the “Get swi Model” button and you should see your model’s code appear on screen:

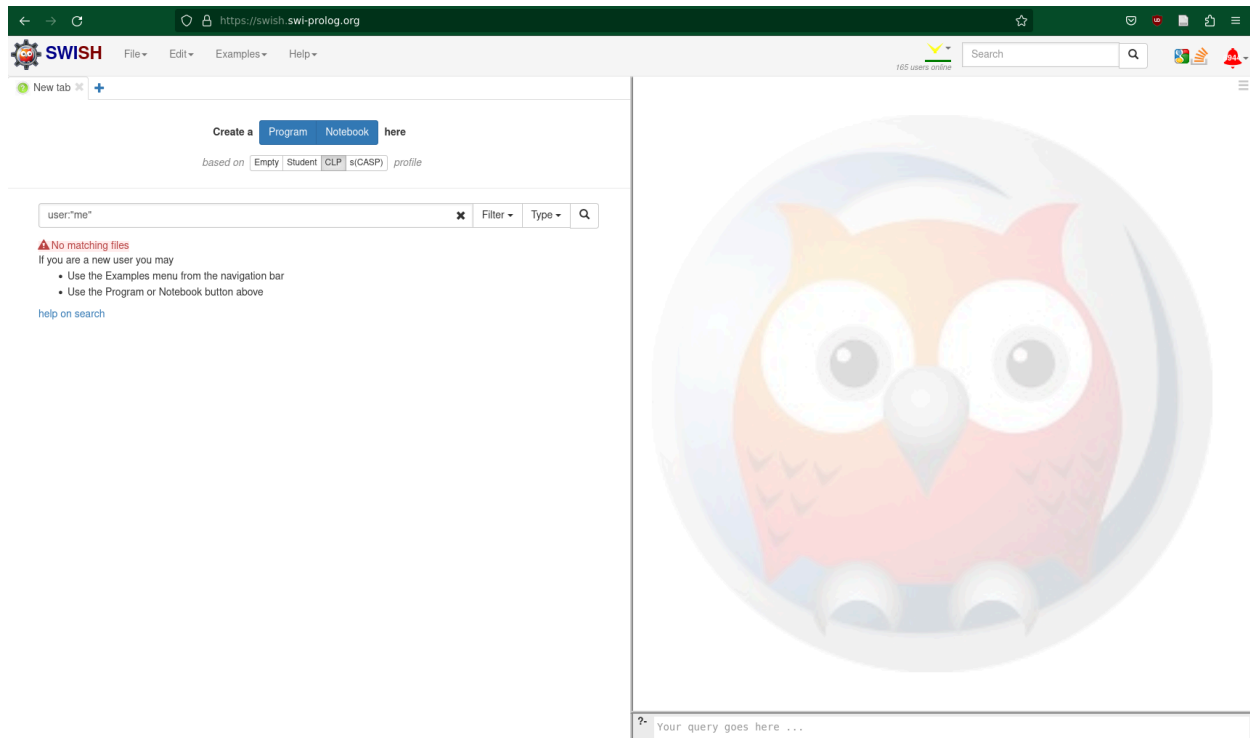


What you see in the text area in the center of the modal is a translation of your model into its constraint logic programming (CLP) representation. These can now be used to perform different analyses of your model.

Part 3: Analyzing the model with its CLP representation

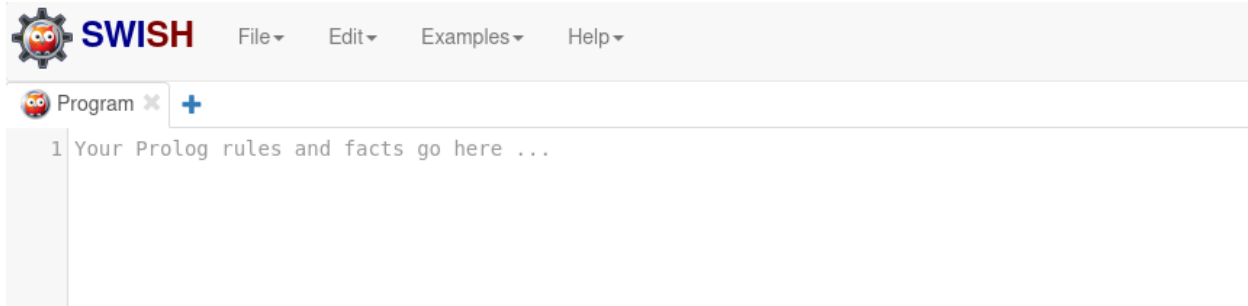
Getting the model into SWI Prolog

We are now going to put these semantics to use. Open a new browser tab, and navigate to <https://swish.swi-prolog.org/> where you should see the following:

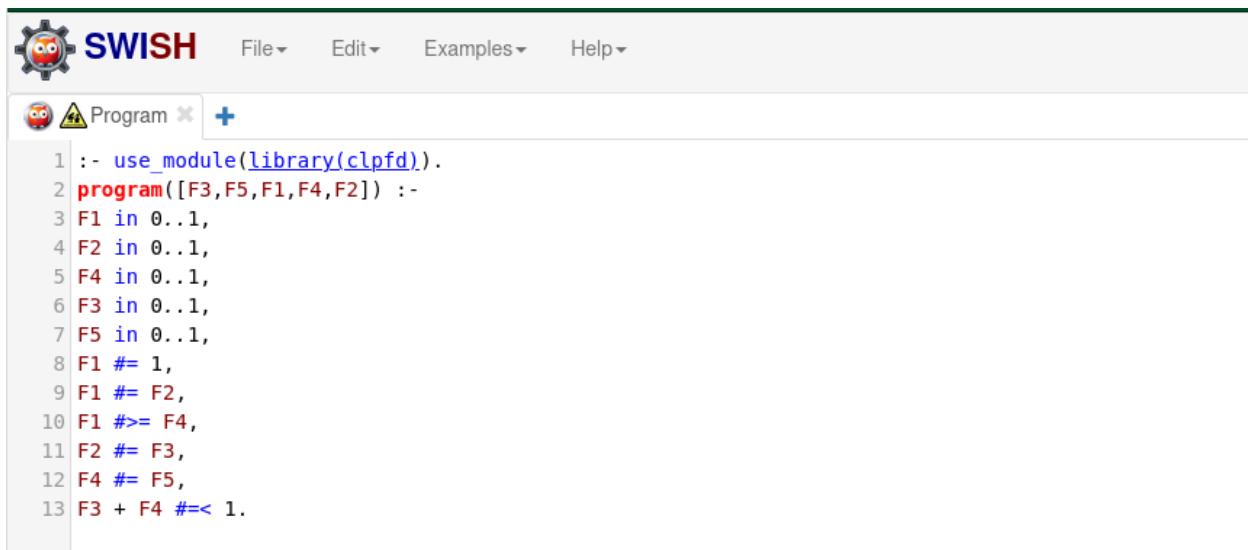


What you see onscreen is an online SWI Prolog playground that allows users to run prolog programs without needing to install anything locally and also to have shared collaborative sessions on the same files. We will be using this online programming environment to run our queries on our model.

Underneath the “Create a Program|Notebook here” heading, set the “based on” selector to “Empty” and click on the “Program” button which should open a new empty editor:



Go back to the VariaMos tab and copy the code corresponding to your model and paste it into the “SWISH” editor:



Now having loaded our program we can examine what it allows us to do. Fortunately, Prolog is remarkably simple and the reading of our program is nearly as it would be if we were reading the actual mathematical notation. Line 1 imports the constraint solver that will allow us to run our program. Line 2 is the declaration of our “predicate” that corresponds roughly to declaring a “function” in a traditional programming language. It states that our program reasons over a list of 5 variables, corresponding to the 5 elements of our model. Lines 3-7 declare that each of these variables is either 0 or 1 (selected or not). Line 8 declares that F1 must be 1, that is, the root must always be present. Lines 9, 11 and 12 are the mandatory relationships between the features meaning that their selection status must match. Line 10 defines the optional relationship obligating the parent to be present if the child feature is selected, but not the opposite. Line 13 defines the exclusion relationship meaning that only one of F3 or F4 can be set to 1 at any time by making their sum always less than or equal to 1.

Basic analysis of the model

We can now execute queries on our program to investigate the properties of our model. The following query will ask for a concrete solution to the constraint program provided:

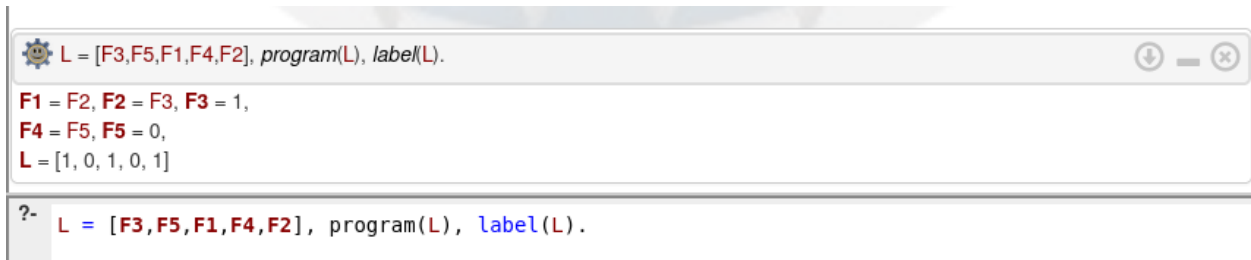
```
L = [F4,F2,F3,F1,F5], program(L), label(L).
```

It first binds the name L to our list of variables for ease of use later and to guarantee that the name and order of variables matches that of the program predicate exactly. It also is convenient for seeing the exact bindings of each variable in the output portion of the SWISH interface. Then it calls our predicate with this list, essentially “applying” all of the aforementioned constraints on these variables such that they are now bound to each other with the relations defined by the model (and therefore the program).

With this done, we can paste this query into the lower right corner of the SWISH editor:



Now we can click on the “Run!” button to obtain a solution to our program and find out what configuration is allowed by our feature model:



As we can see, Prolog tells us that the allowed configuration is F1, F2 and F3 excluding F4 and F5. Intuitively, it seems that our model defines only one valid configuration, but to be certain of this fact we can proceed in multiple ways. First, we may want to check if there exists any other solution. To do this, we can ask Prolog explicitly to calculate all of the solutions that satisfy this program by modifying our query as follows:

```
L = [F4,F2,F3,F1,F5], program(L), findall(L, label(L), Ls), length(Ls, NumberOfSols).
```

This works just like before, but we are doing two additional things, first we are asking prolog to explicitly find us all solutions and collect them as a list of lists with the *findall* predicate and then we are using the *length* predicate to count the number of solutions. If we again introduce this into our SWISH editor, we will observe that it reports to us the following:

```
L = [F3,F5,F1,F4,F2], program(L), findall(L, label(L), Ls), length(Ls, NumberOfSols).
F1 = F2, F2 = F3, F3 = NumberOfSols, NumberOfSols = 1,
F4 = F5, F5 = 0,
L = [1, 0, 1, 0, 1],
Ls = [[1, 0, 1, 0, 1]]
?- L = [F3,F5,F1,F4,F2], program(L), findall(L, label(L), Ls), length(Ls, NumberOfSols).
```

This proves that there exists a single solution for our feature model and that we have a defective feature model, as it denotes a “False” Product Line not allowing any other product.

We can also confirm that the issue is related to feature F4 by explicitly asking our Prolog solver whether there is any solution that would include F4 as follows:

```
L = [F4,F2,F3,F1,F5], F4 #= 1, program(L), label(L).
```

This query is much like our first one, but adds a constraint forcing F4 to be selected. If we run it we will observe the following:

```
L = [F3,F5,F1,F4,F2], F4 #= 1, program(L), label(L).
false
?- L = [F3,F5,F1,F4,F2], F4 #= 1, program(L), label(L).
```

Since Prolog tells us that this query is “false” this proves that indeed there can never be a product containing F4.

Manipulating the model’s formal representation

One of the most powerful aspects of working with these models directly as code is that we can modify the representation to our liking. For instance, there are multiple formalizations that can be assigned to our “Excludes” constraint, while one would normally use (in the case of boolean features) our default constraint:

$$Feature_3 + Feature_4 \leq 1$$

With its equivalent Prolog rendition:

```
F3 + F4 #=< 1
```


We may opt for its more general counterpart that is also valid when features are allowed to be integers, as in the case of “Individual Cardinalities”. In such a case we would use the following constraint:

$$Feature_3 * Feature_4 = 0$$

With its equivalent rendition:

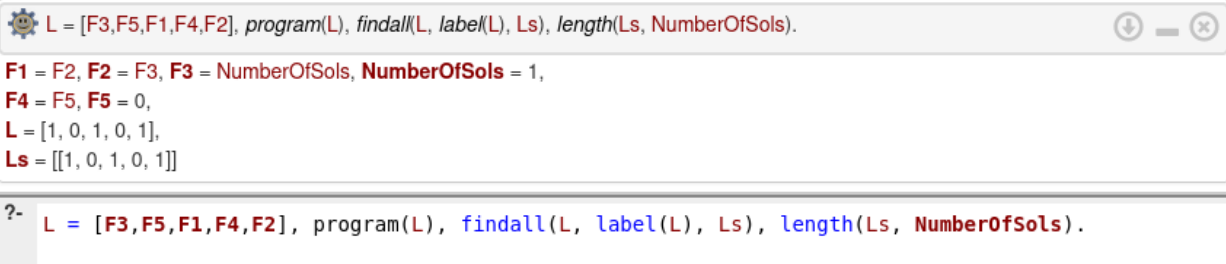

```
F3 * F4 #= 0
```

We can then simply change line 13 of our program in SWISH to obtain the desired program:



```
1 :- use_module(library(clpfd)).
2 program([F3,F5,F1,F4,F2]) :-
3 F1 in 0..1,
4 F2 in 0..1,
5 F4 in 0..1,
6 F3 in 0..1,
7 F5 in 0..1,
8 F1 #= 1,
9 F1 #= F2,
10 F1 #>= F4,
11 F2 #= F3,
12 F4 #= F5,
13 F3 * F4 #= 0.
```

If we then run again our analysis, we observe that the set of solutions remains the same:



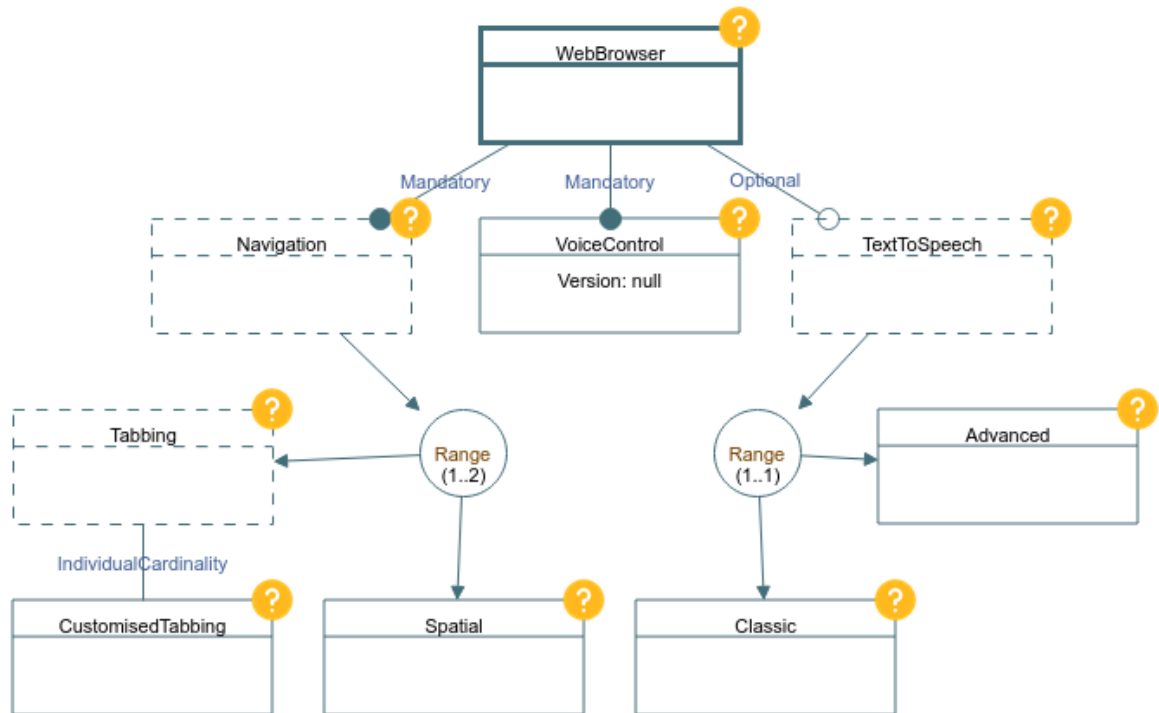
```
L = [F3,F5,F1,F4,F2], program(L), findall(L, label(L), Ls), length(Ls, NumberOfSols).
F1 = F2, F2 = F3, F3 = NumberOfSols, NumberOfSols = 1,
F4 = F5, F5 = 0,
L = [1, 0, 1, 0, 1],
Ls = [[1, 0, 1, 0, 1]]
?- L = [F3,F5,F1,F4,F2], program(L), findall(L, label(L), Ls), length(Ls, NumberOfSols).
```

Part 4: Working with a more complex model

Constructing the model

Having completed the previous portion of this tutorial, now comes the time to analyze larger and more complex models. We will work with a simplified model for accessible web browsers originally proposed in the article: J. Carbonnel, M. Huchard, and C.

Nebut, “Towards complex product line variability modelling: Mining relationships from non-boolean descriptions,” *Journal of Systems and Software*, vol. 156, pp. 341–360, Oct. 2019. Using the techniques described above, construct the following model:



This model has been made available for download and import into VariaMos in the following Link:

https://drive.google.com/file/d/1aP0FPh17BCo8INkC3CpZPnCwf-wPjWJ/view?usp=s_haring

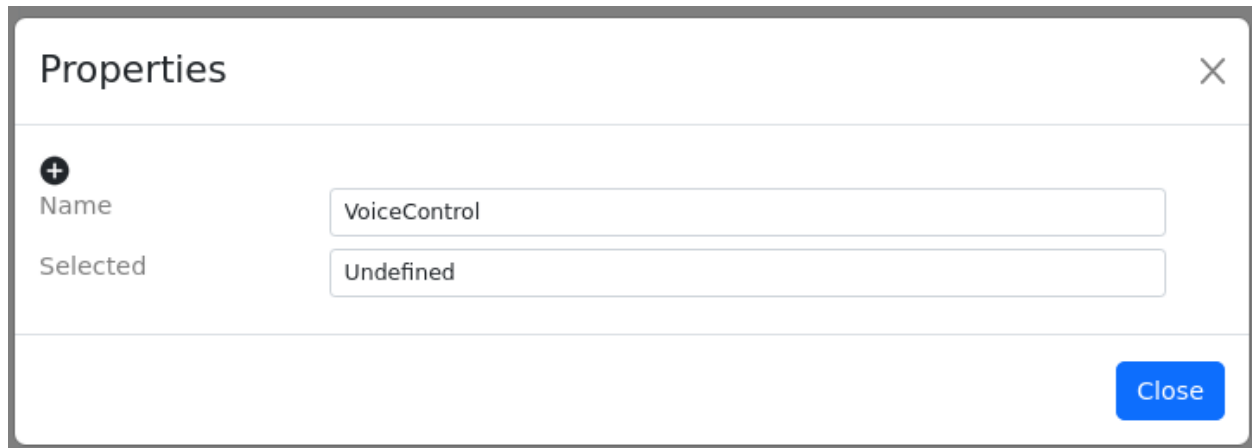
If you import the model, you may proceed directly to Part 5 of this tutorial. If you wish to understand how it is constructed, the additional details building upon the previous tutorial are presented in this section.

Note: The elements that have a “dashed” outline, like “Navigation” are “Abstract Features” (found in the element bar to the right just below the “Root Feature”). They are *functionally and semantically* identical to concrete features, but serve to both better structure and organize the model and remind users that these need a (or several) “Concrete” feature(s) underneath for the model to be well structured (in particular when thinking about each feature’s meaning from a business perspective).

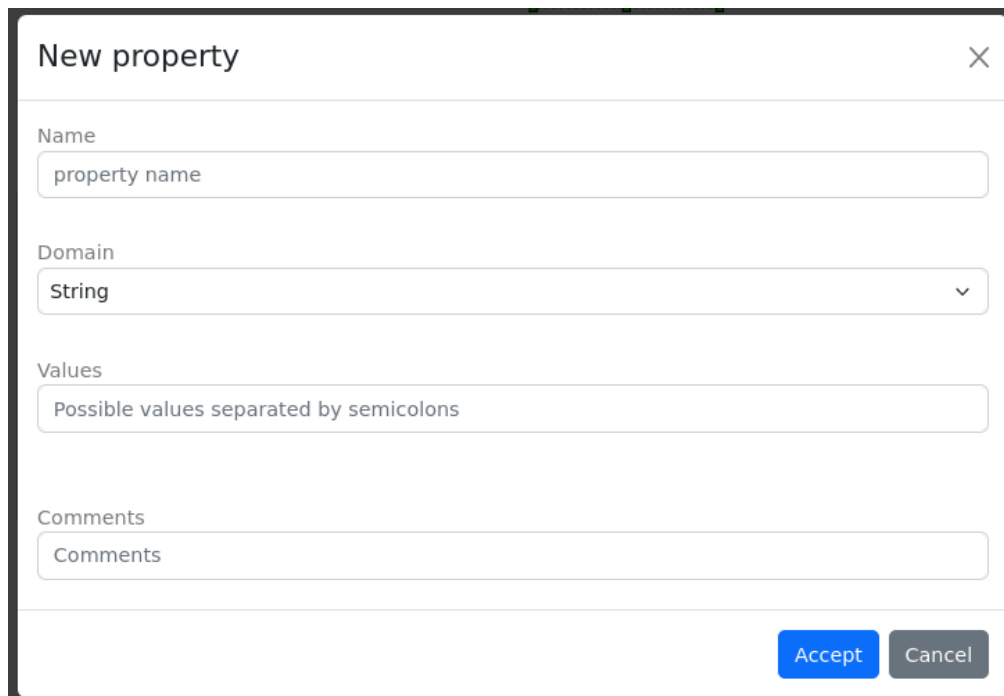
This model utilizes your knowledge coming from tutorial #2. There, string attributes were shown, however, in this case we will introduce the use of “Integer” attributes for features. Though the “Range” elements were already covered in Tutorial #2, we will

nevertheless add some detail on their use in our case. We will explain in a moment how these are used, for now just include them in the model and make sure that the arrows always flow in a downward direction as shown on the diagram.

In order to add the integer attribute to the “VoiceControl” feature, first add the feature and double click on it to inspect its properties.



Next, click on the black “plus” icon above “Name” to add a new attribute. You will see the following screen:



Fill the “Name” field with the name “Version” and change the “Domain” field to integer and then click accept. You should now see the field in the properties modal:

Properties

+

Name	<input type="text" value="VoiceControl"/>
Selected	<input type="text" value="Undefined"/>
Version	<input type="text"/>

Close

Once this is done, you have successfully added a new attribute to the feature. Do note also that the relationship between “Tabbing” and “CustomizedTabbing” has type “IndividualCardinality”, which requires changing the type of the relationship as follows. First create the relationship between “Tabbing” and “CustomizedTabbing” and double click on the relationship itself to show its properties:

Properties

+

Name	<input type="text" value="-"/>
Type	<input type="text" value="IndividualCardinality"/>

Close

Change the type of the relationship to "IndividualCardinality" to reveal the range parameters.

The screenshot shows a 'Properties' dialog box with a close button (X) in the top right corner. On the left, there is a plus sign (+) icon. The dialog contains four input fields: 'Name' with a hyphen (-), 'Type' with 'IndividualCardinality', 'MinValue' which is empty, and 'MaxValue' which is empty. A blue 'Close' button is located in the bottom right corner.

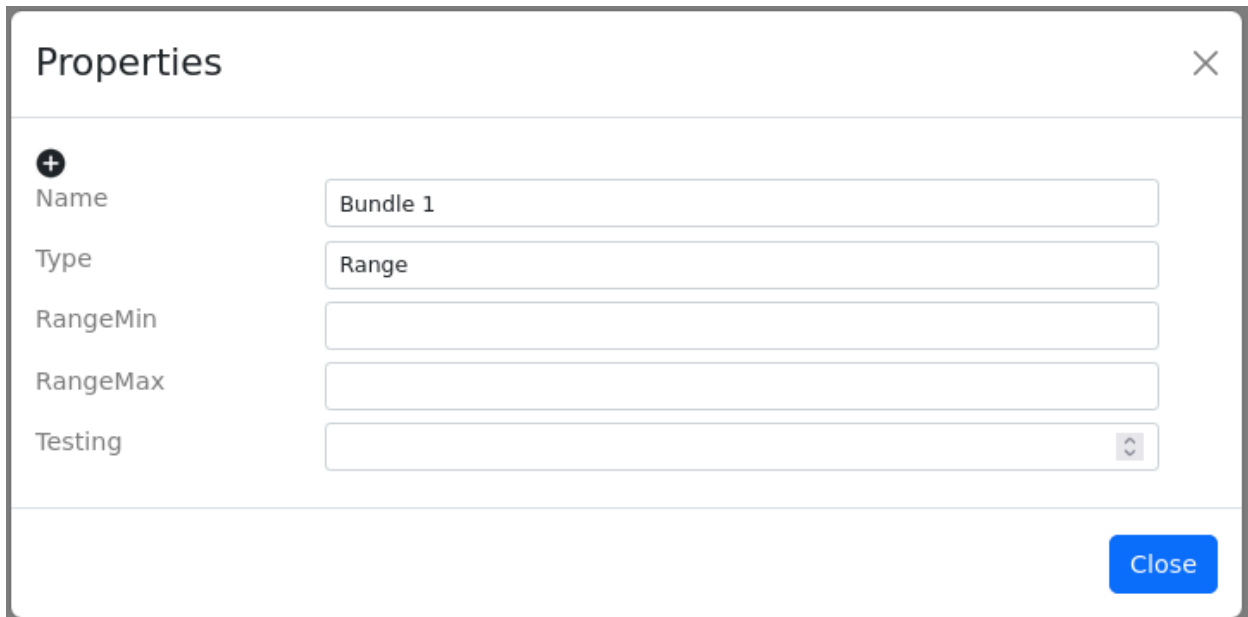
Fill out the “MinValue” with 0 and “MaxValue” with 3 to obtain:

This screenshot shows the same 'Properties' dialog box as above, but with the 'MinValue' field containing the number '0' and the 'MaxValue' field containing the number '3'. All other fields and the 'Close' button remain the same.

This means, intuitively, that the “CustomisedTabbing” feature can be present either as 1, 2, or 3 instances or not at all (0).

The “Range” elements in the model are *VariaMos*’ mechanism of representing “Group cardinalities” in Feature Models. The bundles mentioned above encode all the possible feature groupings, including AND groups (where all the child features must be included), OR groups (where any number or none at all can be included), XOR (where only one of the features must be selected) and user supplied “Ranges” which are the cardinalities mentioned above. To change these bundles into group cardinalities, simply double click on the bundle as you would with a feature and change its “Type” to Range. As with individual cardinalities, you must reopen the modal to see the

“RangeMin” and “RangeMax” properties.



Property	Value
Name	Bundle 1
Type	Range
RangeMin	
RangeMax	
Testing	

Now, for our model above, we must set the range under “Navigation” to 1 to 2, and we must set the range under “TextToSpeech” to 1 to 1 (which could also be an XOR node, but for uniformity we will use a Range as in the original paper). With this we have completed the construction of the basic structure of our model.

We are, however, not done, to have a usable model, we also need additional constraints (following those from the original paper). First, let’s consider the constraint outlined in the original paper where the presence of “Advanced” places a restriction on the value of the Version attribute of the VoiceControl feature. The logical representation of this constraint is as follows:

$$(Advanced = 1) \Rightarrow (VoiceControl::Version \geq 2)$$

Now, in order to encode this constraint and add it to our model, we must add a textual constraint since this is not something that feature models can represent graphically.

VariaMos utilizes the CLIF language, which is a LISP-like language (and an ISO Standard, ISO 24707:2018, available for free here:

https://standards.iso.org/ittf/PubliclyAvailableStandards/c066249_ISO_IEC_24707_2018.zip) for encoding logical formulas. The translation into CLIF of our formula is as follows:

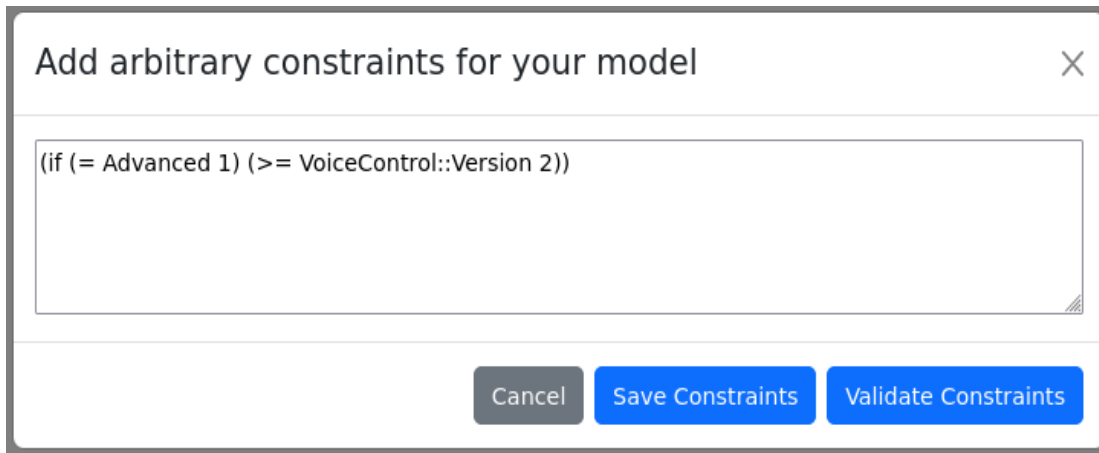
```
(if (= Advanced 1) (>= VoiceControl::Version 2))
```

As can be seen, it is essentially a one-to-one mapping from our formula above, but in “**prefix**” form, where the operators are always put before the operands and where parentheses are used to split each part of the formula.

To include this in our model, we must first open the “arbitrary constraints panel”. To open it, first click on the ink pen icon in the central toolbar (second item from the left).



This will open a new panel where you can add the above constraint:



However, we are not completely done, since we are still missing one important constraint regarding the “Version” attribute of “VoiceControl”, in particular, we haven’t restricted its domain. We must therefore add a constraint that expresses this.

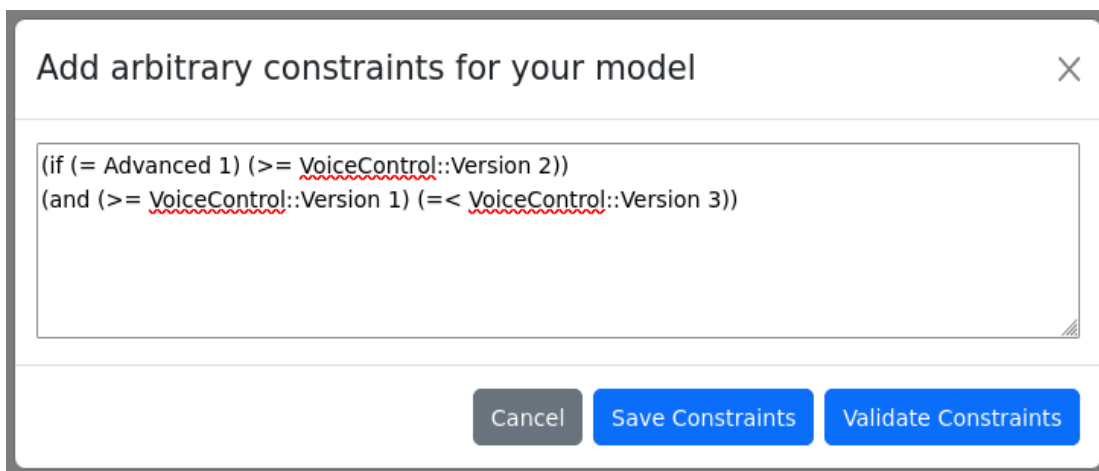
Assuming the available versions are 1, 2, and 3. We can express this as follows:

$(VoiceControl::Version \geq 1) \wedge (VoiceControl::Version \leq 3)$

If we represent this in CLIF, we obtain the following:

```
(and (>= VoiceControl::Version 1) (= < VoiceControl::Version 3))
```

We will therefore end up with the following set of constraints:

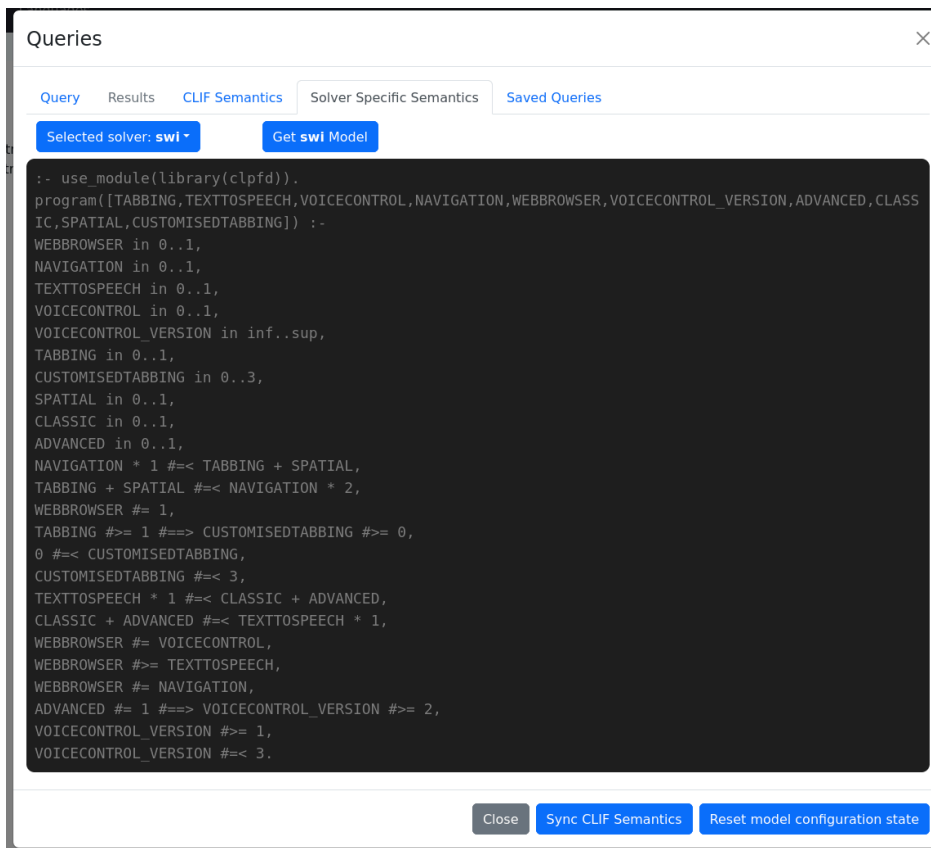


Now, with this complete model, we can now proceed to obtain its concrete semantics just as before.

Part 5: Running queries with the more complex model

Obtaining the model's code

Having constructed our model, we can now tackle the task of utilizing it as we did above to run queries on it. If we repeat the steps from Part 2 of this tutorial we will obtain the following code for our model in SWI prolog:



The screenshot shows a window titled "Queries" with a close button in the top right corner. Below the title bar are four tabs: "Query", "Results", "CLIF Semantics", "Solver Specific Semantics", and "Saved Queries". The "Query" tab is active. Below the tabs are two buttons: "Selected solver: swi" and "Get swi Model". The main area of the window contains a dark background with white text representing Prolog code. At the bottom of the window are three buttons: "Close", "Sync CLIF Semantics", and "Reset model configuration state".

```
:- use_module(library(clpfd)).
program([TABBING,TEXTTOSPEECH,VOICECONTROL,NAVIGATION,WEBBROWSER,VOICECONTROL_VERSION,ADVANCED,CLASSIC,SPATIAL,CUSTOMISEDABBING]) :-
WEBBROWSER in 0..1,
NAVIGATION in 0..1,
TEXTTOSPEECH in 0..1,
VOICECONTROL in 0..1,
VOICECONTROL_VERSION in inf..sup,
TABBING in 0..1,
CUSTOMISEDABBING in 0..3,
SPATIAL in 0..1,
CLASSIC in 0..1,
ADVANCED in 0..1,
NAVIGATION * 1 #=< TABBING + SPATIAL,
TABBING + SPATIAL #=< NAVIGATION * 2,
WEBBROWSER #= 1,
TABBING #>= 1 #=> CUSTOMISEDABBING #>= 0,
0 #=< CUSTOMISEDABBING,
CUSTOMISEDABBING #=< 3,
TEXTTOSPEECH * 1 #=< CLASSIC + ADVANCED,
CLASSIC + ADVANCED #=< TEXTTOSPEECH * 1,
WEBBROWSER #= VOICECONTROL,
WEBBROWSER #>= TEXTTOSPEECH,
WEBBROWSER #= NAVIGATION,
ADVANCED #= 1 #=> VOICECONTROL_VERSION #>= 2,
VOICECONTROL_VERSION #>= 1,
VOICECONTROL_VERSION #=< 3.
```

We can now, just as before, load it into SWISH:


```
dcg-tab x +
1 :- use_module(library(clpfd)).
2 program([TABBING,TEXTTOSPEECH,VOICECONTROL,NAVIGATION,WEBBROWSER,VOICECONTROL_VERSION,ADVANCED,C
3 WEBBROWSER in 0..1,
4 NAVIGATION in 0..1,
5 TEXTTOSPEECH in 0..1,
6 VOICECONTROL in 0..1,
7 VOICECONTROL_VERSION in inf..sup,
8 TABBING in 0..1,
9 CUSTOMISEDTABBING in 0..3,
10 SPATIAL in 0..1,
11 CLASSIC in 0..1,
12 ADVANCED in 0..1,
13 NAVIGATION * 1 #=< TABBING + SPATIAL,
14 TABBING + SPATIAL #=< NAVIGATION * 2,
15 WEBBROWSER #= 1,
16 TABBING #>= 1 #=> CUSTOMISEDTABBING #>= 0,
17 0 #=< CUSTOMISEDTABBING,
18 CUSTOMISEDTABBING #=< 3,
19 TEXTTOSPEECH * 1 #=< CLASSIC + ADVANCED,
20 CLASSIC + ADVANCED #=< TEXTTOSPEECH * 1,
21 WEBBROWSER #= VOICECONTROL,
22 WEBBROWSER #>= TEXTTOSPEECH,
23 WEBBROWSER #= NAVIGATION,
24 ADVANCED #= 1 #=> VOICECONTROL_VERSION #>= 2,
25 VOICECONTROL_VERSION #>= 1,
26 VOICECONTROL_VERSION #=< 3.
27
```

This program, though more complex, includes all of the constraints we've introduced into the model, and, in particular, contains the "arbitrary" constraints we added in the previous part, i.e., those that pose the complex constraint over the "Version" attribute. Do note in particular that our translation treats the attributes in a somewhat special manner, because its name in the program is "VOICECONTROL_VERSION", which allows us to clearly determine that it is related to the "VOICECONTROL" feature, though beyond that it is treated simply as a variable.

We can now run queries just as before on the model, for instance, let's find all the possible solutions to our program with the following (simplified) query:

```
findall(L, (program(L), label(L)), Ls).
```

If we run it, we'll observe that there is quite a considerable number of possible solutions:

```

findall(L, (program(L), label(L)), Ls).
Ls =
[[0, 0, 1, 1, 1, 1, 0, 0, 1, 0], [0, 0, 1, 1, 1, 1, 0, 0, 1, 1], [0, 0, 1, 1, 1, 1, 0, 0, 1, 2], [0, 0, 1, 1, 1, 1, 0, 0, 1, 3], [0, 0, 1, 1, 1, 2, 0, 0, 1, 0],
[0, 0, 1, 1, 1, 2, 0, 0, 1, 1], [0, 0, 1, 1, 1, 2, 0, 0, 1, 2], [0, 0, 1, 1, 1, 2, 0, 0, 1, 3], [0, 0, 1, 1, 1, 3, 0, 0, 1, 0], [0, 0, 1, 1, 1, 3, 0, 0, 1, 1],
[0, 0, 1, 1, 1, 3, 0, 0, 1, 2], [0, 0, 1, 1, 1, 3, 0, 0, 1, 3], [0, 1, 1, 1, 1, 1, 0, 1, 1, 0], [0, 1, 1, 1, 1, 1, 0, 1, 1, 1], [0, 1, 1, 1, 1, 1, 0, 1, 1, 2],
[0, 1, 1, 1, 1, 1, 0, 1, 1, 3], [0, 1, 1, 1, 1, 2, 0, 1, 1, 0], [0, 1, 1, 1, 1, 2, 0, 1, 1, 1], [0, 1, 1, 1, 1, 2, 0, 1, 1, 2], [0, 1, 1, 1, 1, 2, 0, 1, 1, 3],
[0, 1, 1, 1, 1, 2, 1, 0, 1, 0], [0, 1, 1, 1, 1, 2, 1, 0, 1, 1], [0, 1, 1, 1, 1, 2, 1, 0, 1, 2], [0, 1, 1, 1, 1, 2, 1, 0, 1, 3], [0, 1, 1, 1, 1, 3, 0, 1, 1, 0],
[0, 1, 1, 1, 1, 3, 0, 1, 1, 1], [0, 1, 1, 1, 1, 3, 0, 1, 1, 2], [0, 1, 1, 1, 1, 3, 0, 1, 1, 3], [0, 1, 1, 1, 1, 3, 1, 0, 1, 0], [0, 1, 1, 1, 1, 3, 1, 0, 1, 1],
[0, 1, 1, 1, 1, 3, 1, 0, 1, 2], [0, 1, 1, 1, 1, 3, 1, 0, 1, 3], [1, 0, 1, 1, 1, 1, 0, 0, 0, 0], [1, 0, 1, 1, 1, 1, 0, 0, 0, 1], [1, 0, 1, 1, 1, 1, 0, 0, 0, 2],
[1, 0, 1, 1, 1, 1, 0, 0, 0, 3], [1, 0, 1, 1, 1, 1, 0, 0, 1, 0], [1, 0, 1, 1, 1, 1, 0, 0, 1, 1], [1, 0, 1, 1, 1, 1, 0, 0, 1, 2], [1, 0, 1, 1, 1, 1, 0, 0, 1, 3],
[1, 0, 1, 1, 1, 2, 0, 0, 0, 0], [1, 0, 1, 1, 1, 2, 0, 0, 0, 1], [1, 0, 1, 1, 1, 2, 0, 0, 0, 2], [1, 0, 1, 1, 1, 2, 0, 0, 0, 3], [1, 0, 1, 1, 1, 2, 0, 0, 1, 0],
[1, 0, 1, 1, 1, 2, 0, 0, 1, 1], [1, 0, 1, 1, 1, 2, 0, 0, 1, 2], [1, 0, 1, 1, 1, 2, 0, 0, 1, 3], [1, 0, 1, 1, 1, 3, 0, 0, 0, 0], [1, 0, 1, 1, 1, 3, 0, 0, 0, 1],
[1, 0, 1, 1, 1, 3, 0, 0, 0, 2], [1, 0, 1, 1, 1, 3, 0, 0, 0, 3], [1, 0, 1, 1, 1, 3, 0, 0, 1, 0], [1, 0, 1, 1, 1, 3, 0, 0, 1, 1], [1, 0, 1, 1, 1, 3, 0, 0, 1, 2],
[1, 0, 1, 1, 1, 3, 0, 0, 1, 3], [1, 1, 1, 1, 1, 1, 0, 1, 0, 0], [1, 1, 1, 1, 1, 1, 0, 1, 0, 1], [1, 1, 1, 1, 1, 1, 0, 1, 0, 2], [1, 1, 1, 1, 1, 1, 0, 1, 0, 3],
[1, 1, 1, 1, 1, 1, 0, 1, 1, 0], [1, 1, 1, 1, 1, 1, 0, 1, 1, 1], [1, 1, 1, 1, 1, 1, 0, 1, 1, 2], [1, 1, 1, 1, 1, 1, 0, 1, 1, 3], [1, 1, 1, 1, 1, 2, 0, 1, 0, 0],
[1, 1, 1, 1, 1, 2, 0, 1, 0, 1], [1, 1, 1, 1, 1, 2, 0, 1, 0, 2], [1, 1, 1, 1, 1, 2, 0, 1, 0, 3], [1, 1, 1, 1, 1, 2, 0, 1, 1, 0], [1, 1, 1, 1, 1, 2, 0, 1, 1, 1],
[1, 1, 1, 1, 1, 2, 0, 1, 1, 2], [1, 1, 1, 1, 1, 2, 0, 1, 1, 3], [1, 1, 1, 1, 1, 2, 1, 0, 0, 0], [1, 1, 1, 1, 1, 2, 1, 0, 0, 1], [1, 1, 1, 1, 1, 2, 1, 0, 0, 2],
[1, 1, 1, 1, 1, 2, 1, 0, 0, 3], [1, 1, 1, 1, 1, 2, 1, 0, 1, 0], [1, 1, 1, 1, 1, 2, 1, 0, 1, 1], [1, 1, 1, 1, 1, 2, 1, 0, 1, 2], [1, 1, 1, 1, 1, 2, 1, 0, 1, 3],
[1, 1, 1, 1, 1, 3, 0, 1, 0, 0], [1, 1, 1, 1, 1, 3, 0, 1, 0, 1], [1, 1, 1, 1, 1, 3, 0, 1, 0, 2], [1, 1, 1, 1, 1, 3, 0, 1, 0, 3], [1, 1, 1, 1, 1, 3, 0, 1, 1, 0],
[1, 1, 1, 1, 1, 3, 0, 1, 1, 1], [1, 1, 1, 1, 1, 3, 0, 1, 1, 2], [1, 1, 1, 1, 1, 3, 0, 1, 1, 3], [1, 1, 1, 1, 1, 3, 1, 0, 0, 0], [1, 1, 1, 1, 1, 3, 1, 0, 0, 1],
[1, 1, 1, 1, 1, 3, 1, 0, 0, 2], [1, 1, 1, 1, 1, 3, 1, 0, 0, 3], [1, 1, 1, 1, 1, 3, 1, 0, 1, 0], [1, 1, 1, 1, 1, 3, 1, 0, 1, 1], [1, 1, 1, 1, 1, 3, 1, 0, 1, 2],
[1, 1, 1, 1, 1, 3, 1, 0, 1, 3]]
?- findall(L, (program(L), label(L)), Ls).

```

As you can imagine, were the model to be bigger, we would quite easily end up with a gigantic list that would make this analysis illegible and might also consume enormous amounts of memory as it accumulates the solutions into the list. We can overcome these limitations by simplifying our query further and avoiding the accumulation of the concrete solutions and simply determining how many solutions are present by inspecting the generated list which will be filled with anonymous variables and thus occupy very little space in memory with the following query, to which we will also add a timing command to better observe the performance and behavior of our solver:

```
time((findall(_, (program(L), label(L)), Ls), length(Ls, NSols)))
```

We should then observe the following result:

```

time((findall(_, (program(L), label(L)), Ls), length(Ls, NSols)))
15,785 inferences, 0.003 CPU in 0.003 seconds (100% CPU, 5613450 Lips)
Ls =
[_708, _714, _720, _726, _732, _738, _744, _750, _756, _762, _768, _774, _780, _786, _792, _798, _804, _810, _816, _822, _828,
_834, _606, _612, _618, _624, _630, _636, _642, _648, _654, _660, _666, _672, _678, _684, _690, _696, _702, _708, _714, _720,
_726, _732, _738, _744, _750, _756, _762, _768, _774, _780, _786, _792, _798, _804, _810, _816, _822, _828, _834, _840, _846,
_852, _858, _864, _870, _876, _882, _888, _894, _900, _906, _912, _918, _924, _930, _936, _942, _948, _954, _960, _966, _972,
_978, _984, _990, _996, _1002, _1008, _1014, _1020, _1026, _1032, _1038, _1044]
,
NSols = 96
?- time((findall(_, (program(L), label(L)), Ls), length(Ls, NSols)))

```

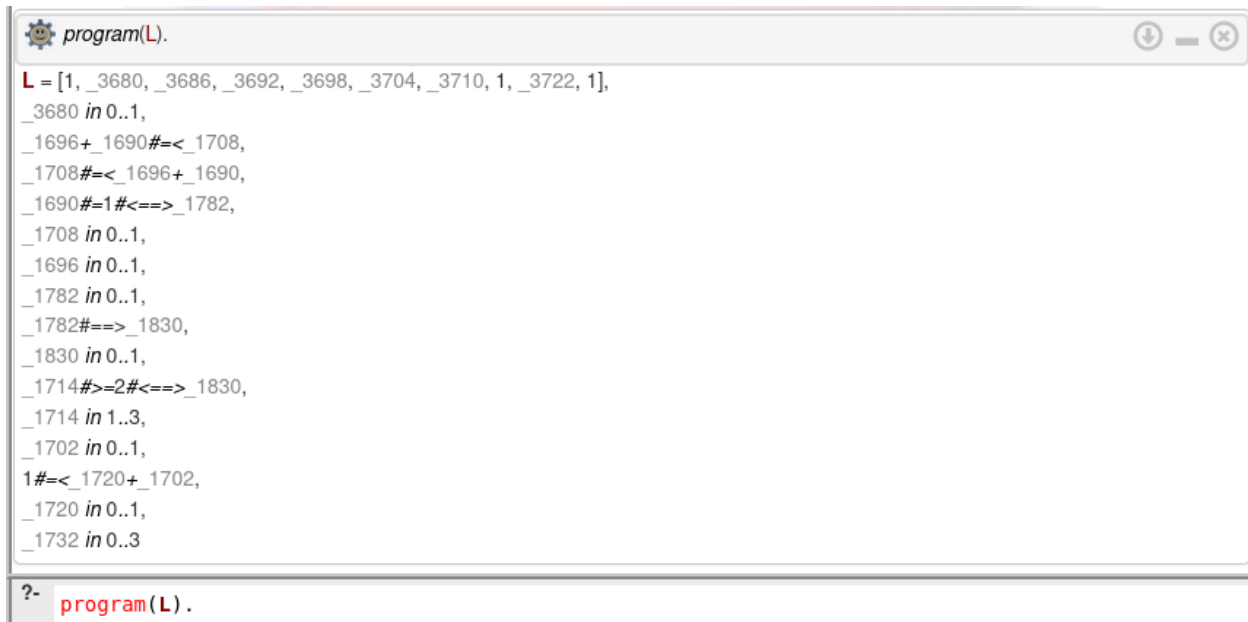
Running more queries on our model

With the knowledge we've gathered so far, and with this non-trivial example, we can begin to pose more interesting queries on our model, for instance, we can check whether particular solutions are allowed by our model. We can also relax certain constraints in order to allow for more solutions.

The very first analysis we can run is for the features that are automatically detected as mandatory by the solver, that is, features that are definitely present in all products and form (part) of the "core" features. We can do this by simply invoking the solver without asking for a particular solution:

```
program(L).
```

This tells us the following:



```
L = [1, _3680, _3686, _3692, _3698, _3704, _3710, 1, _3722, 1],
_3680 in 0..1,
_1696+_1690#<=1708,
_1708#<=1696+_1690,
_1690#<=1#<=>_1782,
_1708 in 0..1,
_1696 in 0..1,
_1782 in 0..1,
_1782#<=>_1830,
_1830 in 0..1,
_1714#>=2#<=>_1830,
_1714 in 1..3,
_1702 in 0..1,
1#<=1720+_1702,
_1720 in 0..1,
_1732 in 0..3
```

```
?- program(L).
```

If we consider the (slightly cleaned up version for legibility since our variables normally overflow offscreen) of our program header:

```
2 program(
3     [WEBBROWSER, ADVANCED, CLASSIC, SPATIAL, TEXTTOSPEECH,
4     VOICECONTROL_VERSION, TABBING, VOICECONTROL,
5     CUSTOMISED TABBING, NAVIGATION
6     ]
7 ) :-
```

We can infer that “WebBrowser”, the root feature, is, unsurprisingly, a core feature. We also can determine that “VoiceControl” and “Navigation” are definitely core features as well. We therefore know that our core features are the full mandatory features as outlined in the diagram. The solver, however, cannot immediately determine if this is the case for others since we would need to check among all the possible solutions if they are never set to 0 and this requires at least some degree of solving. We can however make use of the solver to do this analysis by first slightly modifying our program with the following directives that will allow us to run more complex queries by giving us access to additional Prolog predicates:

```

1 :- use_module(library(clpfd)).
2 :- use_module(library(yall)).
3 :- use_module(library(apply)).
4 :- use_module(library(apply_macros)).
5 program(
6     [WEBBROWSER,ADVANCED,CLASSIC,SPATIAL,TEXTTOSPEECH,
7       VOICECONTROL_VERSION,TABBING,VOICECONTROL,
8       CUSTOMISEDTABBING,NAVIGATION
9     ]
10 ) :-

```

We can use this complex query to calculate which are core features of the product line and those that are variable:

```

findall(L, (program(L),label(L)), Sols), length(Sols, NSols),
transpose(Sols, TSols), maplist([Vs,R]>>(include([X]>>(X #=
0),Vs,V0s),length(V0s,R)),TSols,VariableFeats).

```

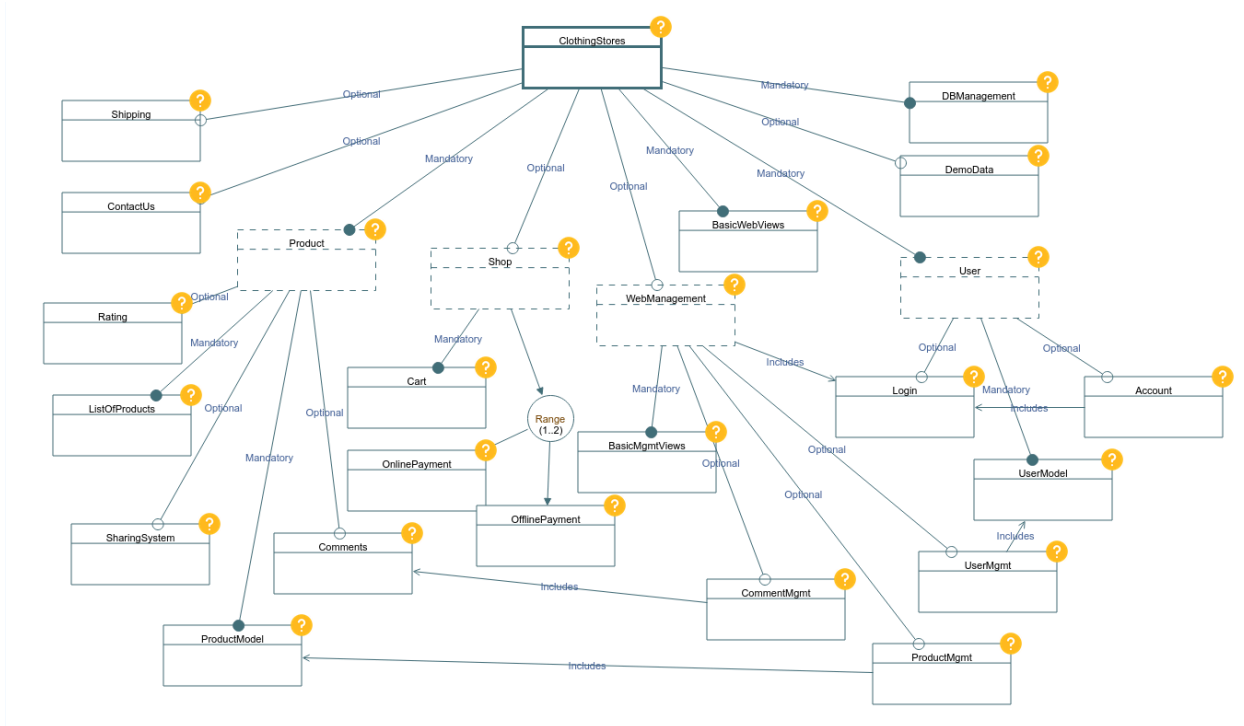
This query is designed such that it will allow us to determine or calculate how many times a feature appears as deselected within the set of solutions. The first part of the query, “*findall(L, (program(L),label(L)), Sols), length(Sols, NSols)*”, as before, serves to calculate the solutions and their number, aggregating them all into the list *Sols*. The second part of our query introduces the additional complexity necessary to perform these more complex calculations, in particular, we calculate the transpose of the list of solutions, giving us therefore lists collecting all the assignments for each of the variables in each of the solutions. We then use the *maplist* meta-predicate on each of these lists where we drop every non-zero entry and then count the number of remaining entries. This gives us then a set of lengths bound to the *VariableFeats* variable that will be a list, in the same order as the original variables, that encodes how many times it has been bound to 0, i.e., deselected. If we run it we will observe the following:

Part 6: Practical exercise

Link to the questionnaire: <https://forms.gle/Di4K5U8o32mfANHi9>

With all of the experience you’ve gained thus far, perform the following:

1. Load the model you completed as part of the previous tutorial’s practical portion into VariaMos, though, for simplicity’s sake, we will not be making use of the attributes within the features. Your model should resemble the following model:



For your convenience and in case you cannot remove the attributes, a version of this model is provided to you in the following link:

https://drive.google.com/file/d/1q72bXro_qKp9ZIWxgCN4W3k_FIGer1qp/view?usp=sharing

2. Utilize VariaMos’ mechanisms for obtaining your model’s representation as code.
3. Insert the code into SWISH as described above.
4. Find a solution.
5. Find 30 solutions.
6. Find the total number of solutions to the model.
7. Test 3 different solutions defined by you.
8. Find the “core” features of the model.

9. Find the “variable” features of the model.