Tutorial #4 - Automated Verification of Models

Introduction

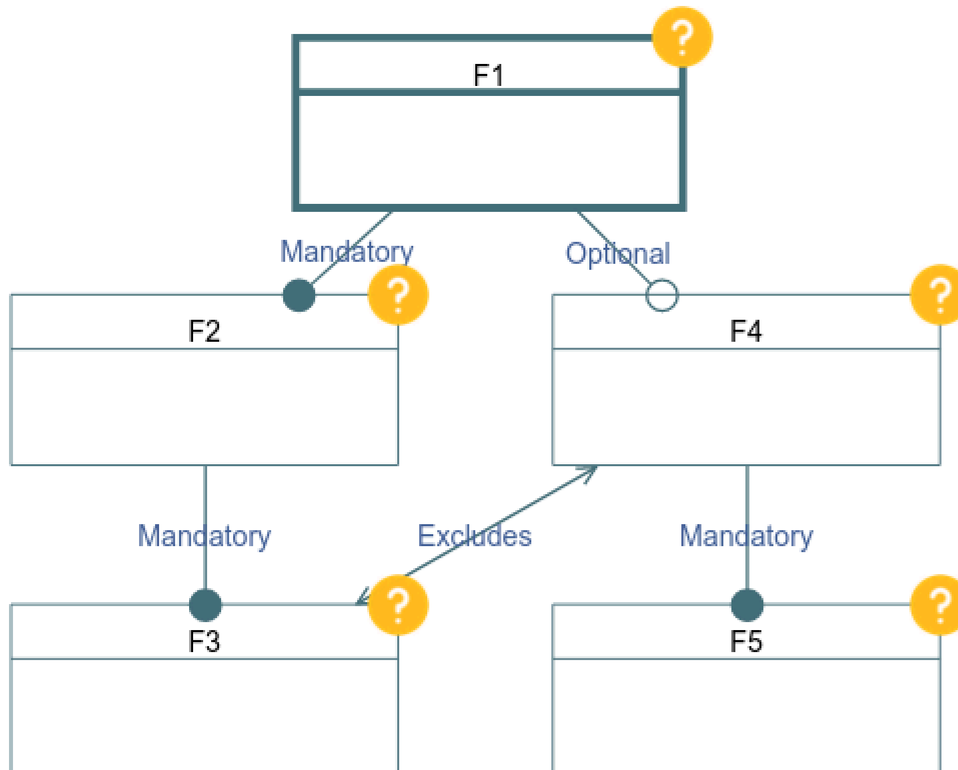
This tutorial aims to demonstrate the mechanisms for automated analyses of Variability Models through the use of the PLEIADES back-end. This back-end service will allow you to perform queries on your system using a JSON-based query specification system. In this tutorial you will learn the basic operation of the system and how it interacts with the VariaMos editor.

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 the 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.
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: Running verification queries on the model

Background - Queries

Our goal is to find defects that may exist in the model. Though for such a simple model, they can be found through manual inspection, this does not scale to large models that may have hundreds of features or more. To do this, we must make use of automated analysis techniques akin to those illustrated in Tutorial #3. That being said, it is not necessarily simple or evident how to formulate the requests as illustrated in that tutorial. More specifically, they require having an understanding of the underlying solver's API and how to control its execution.

To remedy this, we will make use of the capabilities incorporated into the PLEIADES VariaMos module, which extends the translation functionality presented in Tutorial #3 to allow one to automatically run different queries on the models constructed within VariaMos. These queries are analogous to the "queries" run within SWI-Prolog in the earlier tutorial in that they provide a specification of the reasoning task to be performed by the inference engine. The fundamental difference with the queries that we will now present is that our new query system presupposes no knowledge of the underlying programming environment and that the queries are thus formulated at a higher level that corresponds only to the verification task at hand.

The query system we have formulated is based on JSON specifications that allow encode the reasoning request (query) that one wishes to perform on the currently selected model. In addition, VariaMos has been tightly integrated with the PLEIADES reasoning service, allowing for feedback to become immediately visible on the user interface, thus greatly facilitating the tasks at hand.

The query language relies on JSON documents that follow the following schema:

```
{
 "solver": "<The constraint solver you want to use>",
 "operation": "<The basic reasoning operation (SAT check or getting solutions)>",
 "operation_n": "<Optional parameter specifying how many solutions to check for",
 "optimization_target": "<Optional parameter defining the optimization variable>",
 "optimization_direction": "<Optional parameter defining minimization/maximization>",
 "iterate_over": [<An optional list of specifications for iterative operations>]
}
```

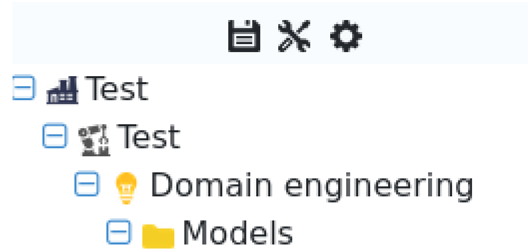
The ***"solver"*** field currently allows for the use of three different systems: Gecode (through the MiniZinc language and programming environment, denoted as "minizinc" in the JSON), SWI-Prolog (denoted as **"swi"**), and the Z3 solver (denoted as **"z3"**). Their selection is based on both the performance of each on the given types of constraints for your language and the types of operations that are required. In particular, Gecode (through minizinc) does not currently support some operations, and while z3 is by far the fastest solver, hasn't been fully integrated, and hence is untested and not fully featured, it has limited support for some operations and characteristics, in particular enumerating many solutions quickly and dealing with attributes.

The **"operation"** field encodes the basic reasoning operation to be performed. These range from satisfiability checks (denoted as **"sat"**), finding a concrete solution (denoted as **"solve"**), finding *n* solutions (denoted as **"nsolve"**, which also needs to be accompanied by the **"operation_n"** parameter specifying, at most, how many solutions are to be returned), and solving an optimization problem (denoted as **"optimize"**, which must be accompanied by the **"optimization_target"** parameter which defines the variable over which the optimization problem will be defined, and by the **"optimization_direction"** parameter which can be either **"min"** or **"max"** to determine whether it is a maximization or minimization problem).

The **"iterate_over"** parameter serves primarily to determine how to run queries that require iterating over elements of the model. For example, determining which features are "dead" in a feature model is a matter of checking whether each of the features can be set to selected in a configuration: otherwise they are "dead". There exist other checks that make use of this parameter that we will explore as we move through this tutorial.

Writing simple queries

We will now learn how to actually write a query to reason automatically on our model. Begin by clicking on the “wrench and screwdriver” icon above the project explorer:



You should now see the following modal (your version may be without the “Translator Endpoint” field). If you are using docker locally you must use the default endpoint shown:

Queries

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

Translator Endpoint

http://localhost:5000/query

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

Query

Enter Query Name

Save Query

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

We’ll begin with the most simple of analyses, checking whether the model has any solutions at all. This will allow us to observe the query language in action.

To do this, we first write the following query in the “Query” text box:

```
{
 "solver": "minizinc",
 "operation": "sat"
}
```

This simple query asks the back-end for two things:

- To use Gecode (through MiniZinc) as the solver.
- To do a “satisfiability” check on the model, that is, check for the presence of at least one solution that satisfies the constraints from the model (this is the

"operation" field in the JSON). The **"sat"** keyword is our encoding for the simple satisfiability check operation. We'll explore the other options available as we move through the tutorial.

Your query should then look as follows in the modal:

Queries

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

Translator Endpoint

http://localhost:5000/query

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

Query

```
{
  "solver": "minizinc",
  "operation": "sat"
}
```

Enter Query Name

Save Query

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

Click on the "Submit Query" button in order to run the operation and switch to the "Results" tab in the modal.

Queries

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

Solution 0	SAT

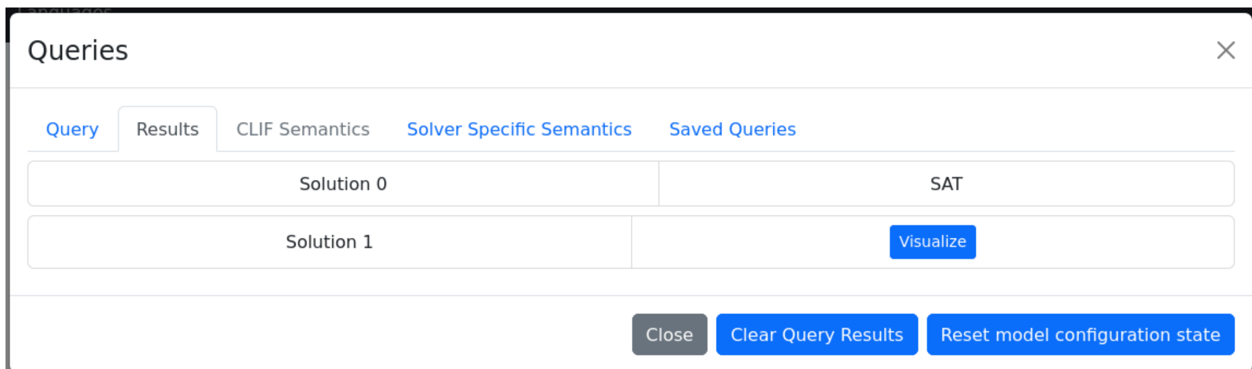
Close | Clear Query Results | Reset model configuration state

As can be seen, the first result (numbered as "Solution 0") tells us that the model is indeed "SAT", or, in other words, that there is a solution for the mode (in verification terms, it means that ***the model is not void***). We, of course, might prefer to see the solution directly in the modeling environment than trying to reconstruct it from the output of the solver. Since this operation is fundamentally different, namely, it involves obtaining a concrete solution, we must use the following query:

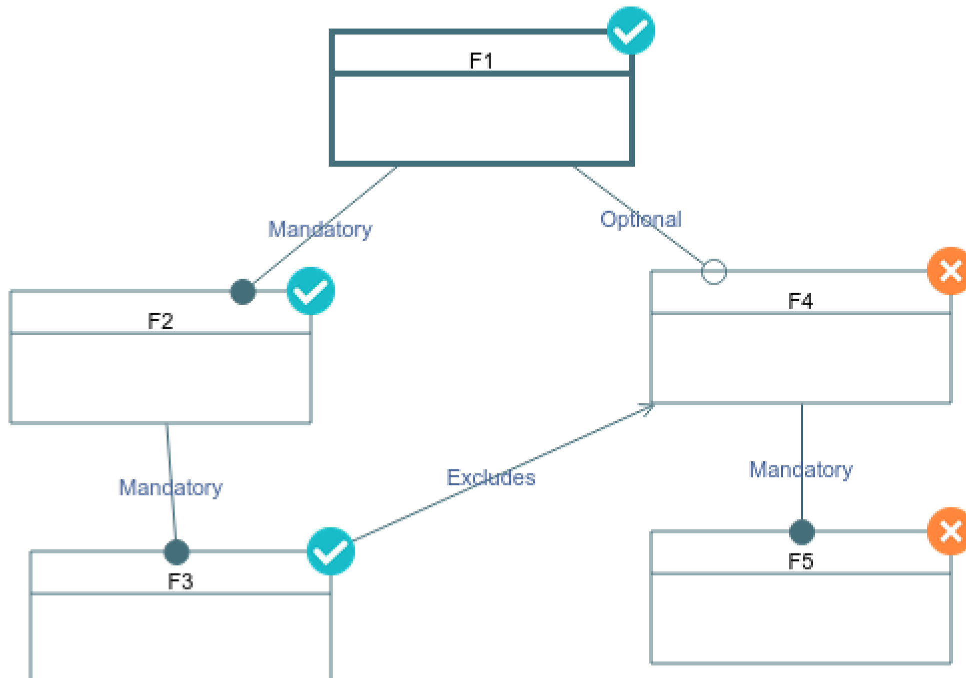
```
{
 "solver": "z3",
```

```
 "operation": "solve"
}
```

As can be seen from the code above, this query is quite similar to the previous one, but we are asking for a concrete solution instead of just its existence (which involves changing the operation from **“sat”** to **“solve”**). In addition and to illustrate the flexibility of this approach, we have also opted to utilize the Z3 solver instead of MiniZinc, though their behavior is and should be ultimately identical. If one inputs this query and runs it, one will observe the following in the “Results” tab:

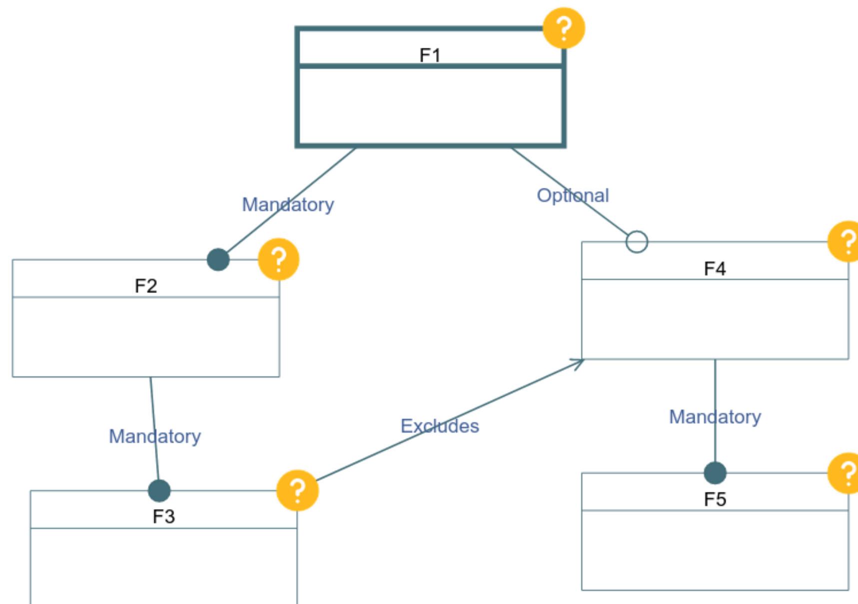


This new result can then be accessed directly from the application by clicking on the “Visualize” button, closing the modal and reselecting the model from the explorer on the left side to reveal the modified model:



This overlays the solution directly on top of the model. This coincides with the solution one would find manually but required no direct manipulation of the solver.

Once this has been done, we might want to check for other solutions or otherwise manipulate the model but since it has had the solution overlaid, we can reset the model to its original state by reopening the “Query” modal and clicking on the “Reset model configuration state” button which will remove the selection from the model.



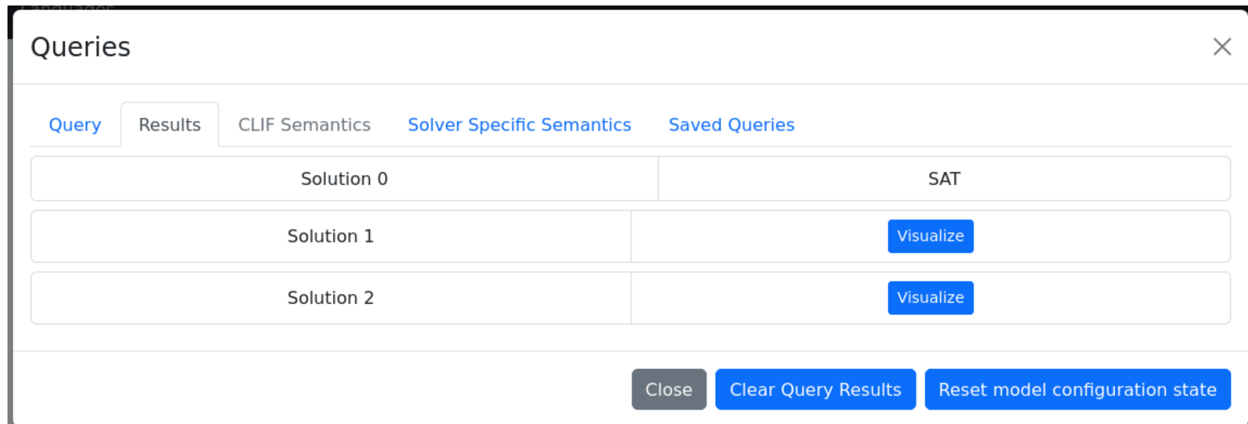
With this done, let’s now check for other solutions by asking our solver to produce a larger set of solutions. To do this, we modify our query slightly into the following:

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

This exemplifies another operation available in our query language which is asking for a set of solutions instead of a single one. To do this we change the **“operation”** to **“nsolve”**, meaning “solve n times”. This operation must be accompanied by another parameter, **“operation_n”,** that tells the solver to find (at most) n solutions. The point about “at most” is worth highlighting, since there are no guarantees that the requested number of solutions do exist. What the solvers will try to do is solve up to the requested amount and return as many solutions as are found. For illustrative purposes we also utilize the **“swi”** solver backend in this case (to be clear, we change the solver

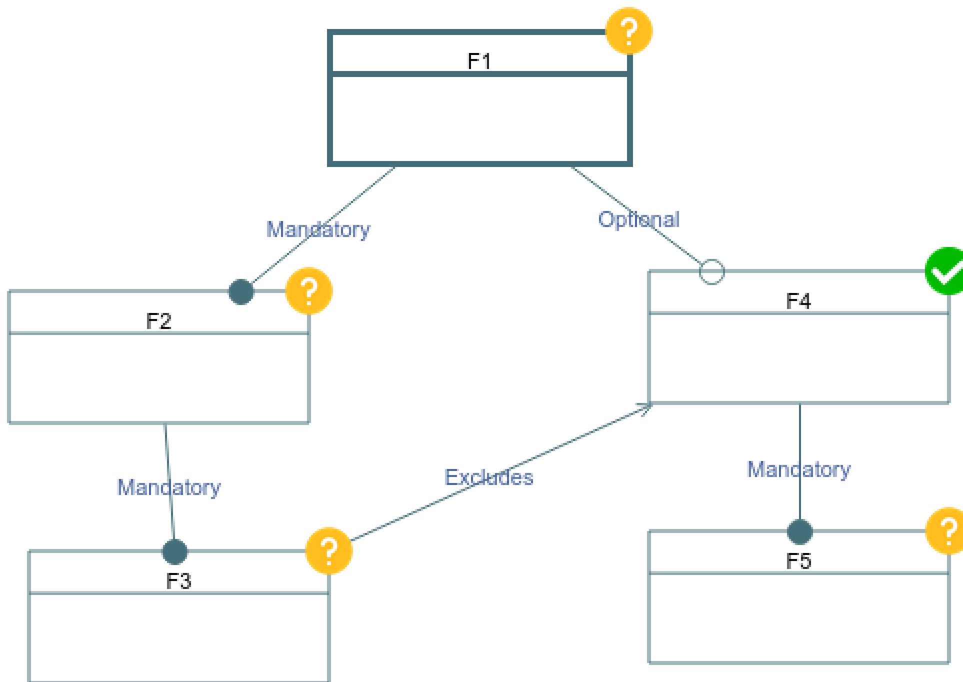
to illustrate the flexibility of the tool). Moreover, this corresponds to one of the semantic checks, that is, checking for a *false product line*, which implies simply determining that at least two solutions exist, whose manual operation we covered in the earlier tutorial.

If we input this new query and run it, we will observe the following results:



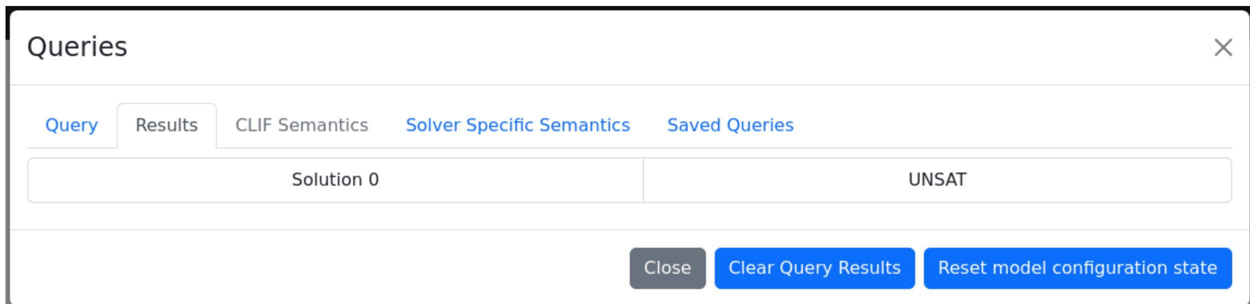
Since we obtain exactly the same response, this illustrates that there is indeed only a single solution and we are sure therefore that no other solution can exist in this model, and therefore this has failed the check for whether this is a ***false product line***.

This leads us naturally to the final part of the analysis which relates to determining the causes of this problem. As one can probably determine by manual inspection, the problem seems related to the "excludes" relation between F3 and F4. Since we saw that F4 is not included in the solution, let's try to see if selecting it leads to problems. Let's then check for problems related to the "F4" feature manually. Click on the "Question Mark" icon on top of Feature F4 util it displays a green check mark as shown below:



This means that we have “set” our feature as “Selected”, meaning that it must appear in the solution as selected (True or 1).

We can now rerun our query asking for a single solution (with any of the solvers) and we should observe the following result:



As can be seen, the model is unsatisfiable and no solution can therefore be found, so we know that ***this feature is “Dead”***, that is, cannot be included in any solution. This, however, is not the ideal way of proceeding since we would need to do this analysis manually for every feature we suspect of being defective. What we should do, instead, is run a single query that will perform this analysis automatically for us. To clean up our results tab, first click on “Clear Query Results” to reset the saved solutions. Next, we will use the following “iterative” query specification:

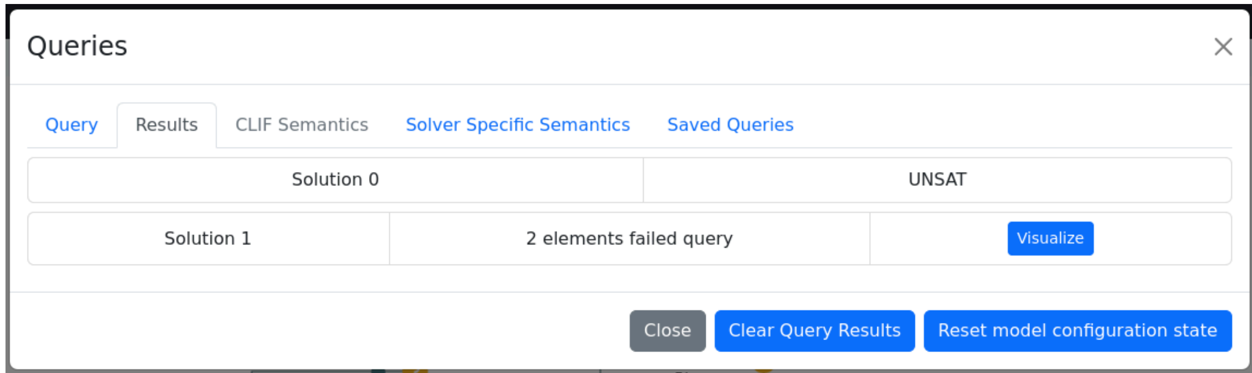
```
{
 "solver": "minizinc",
 "operation": "sat",
```

```
 "iterate_over": [
   {
     "model_object": "element",
     "object_type": ["ConcreteFeature"],
     "with_value": 1
   }
 ]
}
```

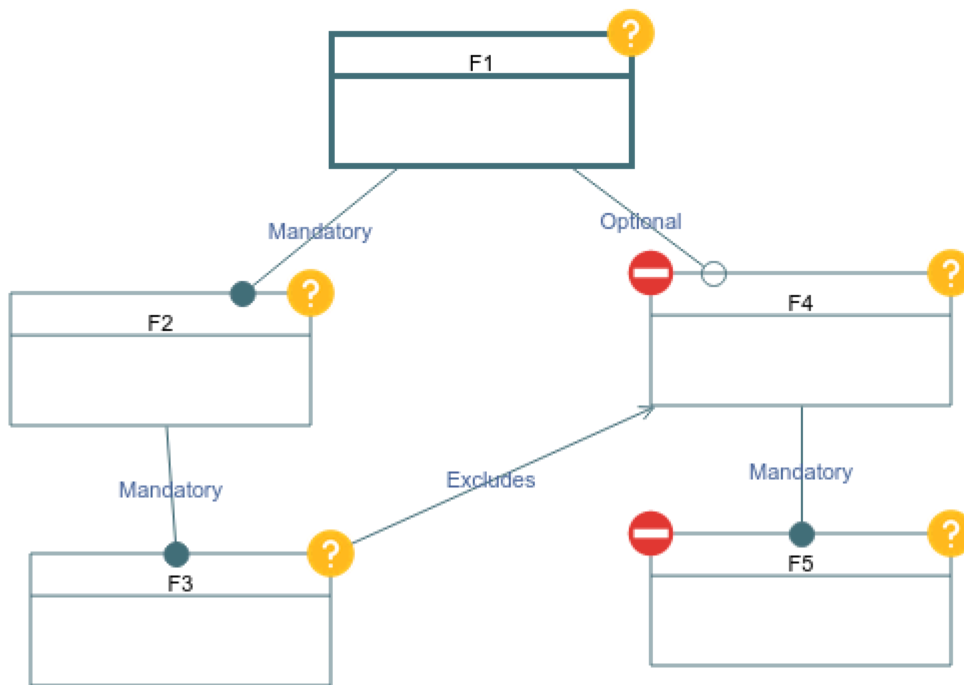
Before we run it, it is important to remind ourselves of two things: first, that checking for a dead feature in Feature Models is essentially determining whether a model is satisfiable if a given feature is set to selected; and, second, that setting a feature to selected, in terms of its semantics, is saying that the feature's associated variable is equal to 1.

As can be seen, it is very similar to the satisfiability query we have already done before, but includes the additional field **"iterate_over".** What we mean precisely by iteration is that we will run the operation multiple times, once for each element in the model that matches the specification for the iteration. The **"iterate_over"** portion of the specification takes a list with objects determining which model elements meet these criteria, somewhat similar to a "select … where" query in SQL. In this particular case we are telling the system that we want to iterate through the **"elements" (denoted as "model_object" : "element" in the spec)**, that is, the nodes of the graph. We also add the parameter **"object_type"** which filters these elements depending on their type. This model has two types of nodes, a root feature and several concrete features. Since we know the root is always included no matter what, we are only interested in the concrete features. This **"object_type"** filter takes in a list of types to iterate over, which in this case has only the **"ConcreteFeature"** type since it is the only thing that interests us. Finally, in the specification, we add the **"with_value"** parameters, which tells the iteration specification what it should do with the elements it will iterate over. In this case, it will, as mentioned above, the variable associated with each feature to 1. (**N.B.** The system currently only accepts integers in this parameter.)

If we run the query, we will observe the following result, telling us that two elements ran into problems with this iterative query:



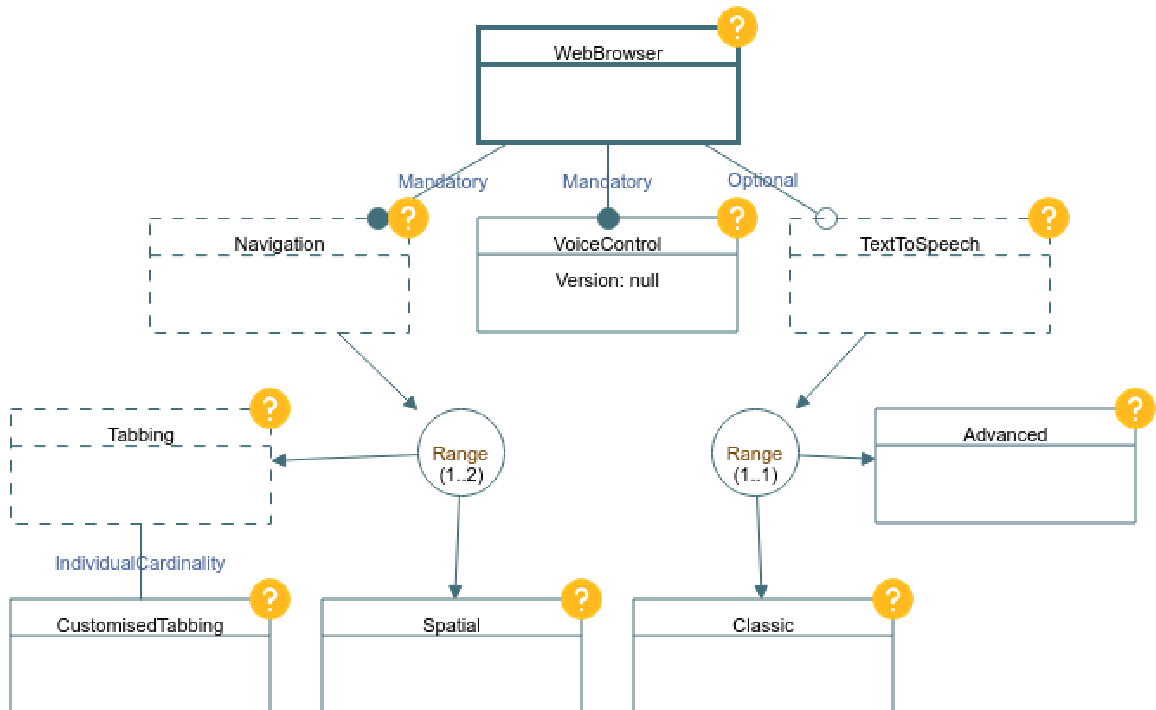
Now click the visualization button, close the modal, and you should see the model change into the following:



These one-way signs now indicate that ***both features F4 and F5 are "Dead"*** and can never be selected in any configuration since they failed the satisfiability check. In the general case, they will be applied to any element whose iteration leads to UNSAT. With this, we see that with considerably less effort we have managed to perform the same reasoning operations as before to determine flaws in this model, and even found that given the mandatory relationship between F4 and F5, its "dead" status carries down to it, without needing to manually check each variable.

Part 3: Scaling up to a larger model

We will now be using the more complex model from Part 4 of Tutorial #3.



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=sharing

Let's now run some analyses on this model. To begin, let's simply check if a solution exists. To do so, we can simply use our simple query:

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

If we run this query we will observe the following:

Queries

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

Translator Endpoint

http://ec2-3-130-187-131.us-east-2.compute.amazonaws.com:5000/query

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

Query

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

Enter Query Name

Save Query

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

Queries

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

Solution 0	SAT

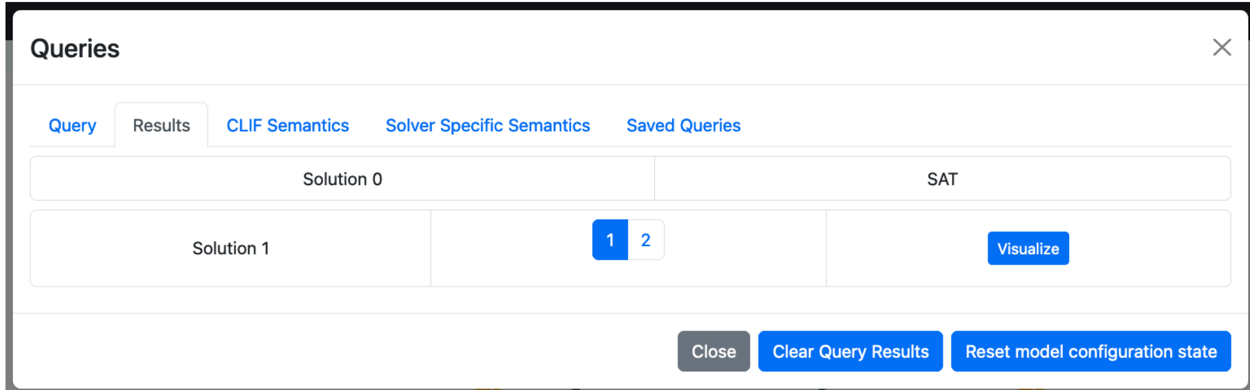
Close | Clear Query Results | Reset model configuration state

This implies, therefore, that the model does indeed have a solution and is not "void". In contrast with Tutorial #3, there was no need to extract the code corresponding to the model, nor use SWISH to run the analysis. With the simple query above, we can ascertain that there is indeed such a configuration.

As in the earlier portion of the tutorial, we can also check that this is not a false product line by simply asking for two products with the very same query we used before.

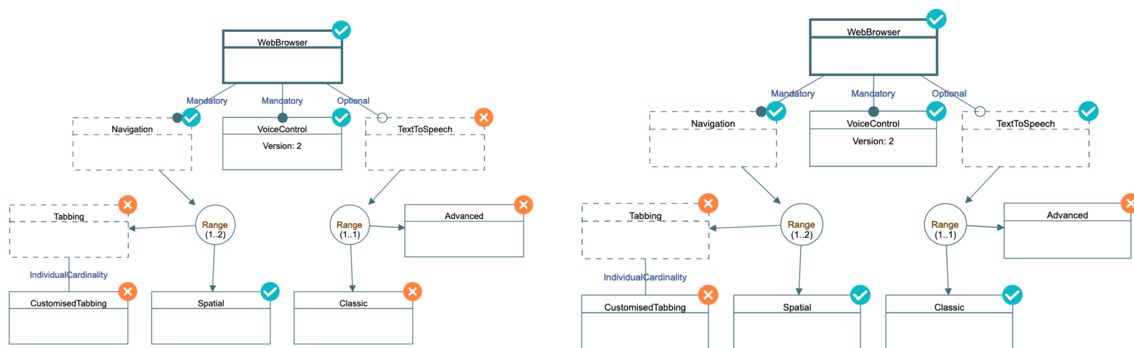
```
{
 "solver": "swi",
 "operation": "nsolve",
 "operation_n": 2
}
```

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



This shows that there are indeed two possible solutions to our problem, and therefore two possible products, which implies that ***this is not in fact a "false" product line***.

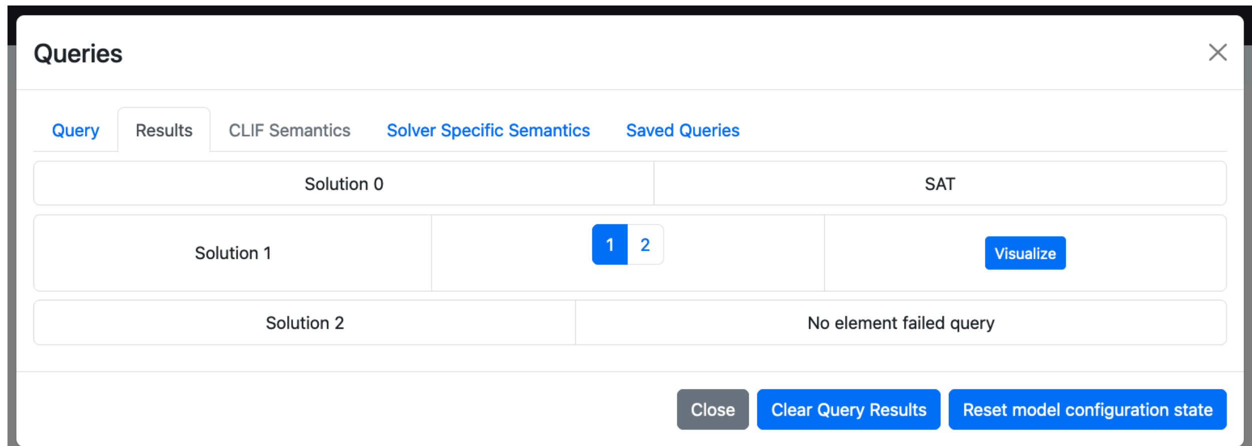
We can use the selector in the middle portion of the result to select which of the solutions we wish to visualize, which allows us to see the following:



As can be seen, two distinct solutions are made available to us directly on the interface, without needing any additional interaction with the solvers to obtain them. Let's continue with our analysis by examining our model for any ***potential dead features*** with the following query:

```
{
 "solver": "minizinc",
 "operation": "sat",
 "iterate_over": [
   {
     "model_object": "element",
     "object_type": ["AbstractFeature","ConcreteFeature"],
     "with_value": 1
   }
 ]
}
```

It is worth noting that the primary modification between this query and the one used on the model from Part 2 is that we have added "AbstractFeature" to the list in "object_type"; the reason for this is simple: we need to be able to also check elements of this type, and it suffices to simply add the element type to the list. Running this query produces the following result:



In contrast to the model examined in Part 2 of this tutorial, no element in the model failed the query, therefore telling us that ***there are no dead features*** in this model.

The key takeaway from this is that the system is simple enough that the same set of queries for the simple model are just as capable as those for the more complex model and so it goes as the model grows.

Part 4: 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 used for the previous tutorial's practical portion into VariaMos.

For your convenience, this model is provided to you in the following link:
https://drive.google.com/file/d/1q72bXro_qKp9ZlWxgCN4W3k_FIGer1qp/view?usp=sharing

2. Utilize the query system to check that the model is indeed satisfiable using "swi" as your solver.
3. With "swi" obtain a configuration for your system.
4. With "swi" check whether the model constitutes a false product line.
5. With "swi" check whether there exist dead features in your model.
6. With "swi" check whether there exist false optional features in your model.