CARYA

Malleable VIs

Wiebe Walstra
Carya Automatisering, Delft

CLA, Champion, alliance partner, etc..

There since at least 2013. Offical in 2016 I think, but only fully functional in 2017SP1, fully supported in 2018. Still using the beta for this demo.

Feel free to interrupt for questions.

Let’s start with the introduction…

Introduction: malleable

[mal-ee-uh-buh-l] **[măl’ē-ə-bəl]**

Adjective

- **able to be worked, hammered, or shaped under pressure or blows without breaking**

Synonyms

- **Impressionable, adaptable, mouldable, flexible, pliable, tractable**

Origin

- **from Latin *malleus*: hammer**

This is what the dictionary has to say. I think most people say [mal-uh-buh-l], not [mal-ee-uh-buh-l]…

Introduction: malleable

[mal-ee-uh-buh-l] **[măl’ē-ə-bəl]**

Adjective

- **able to be worked, hammered, or shaped under pressure or blows without breaking**

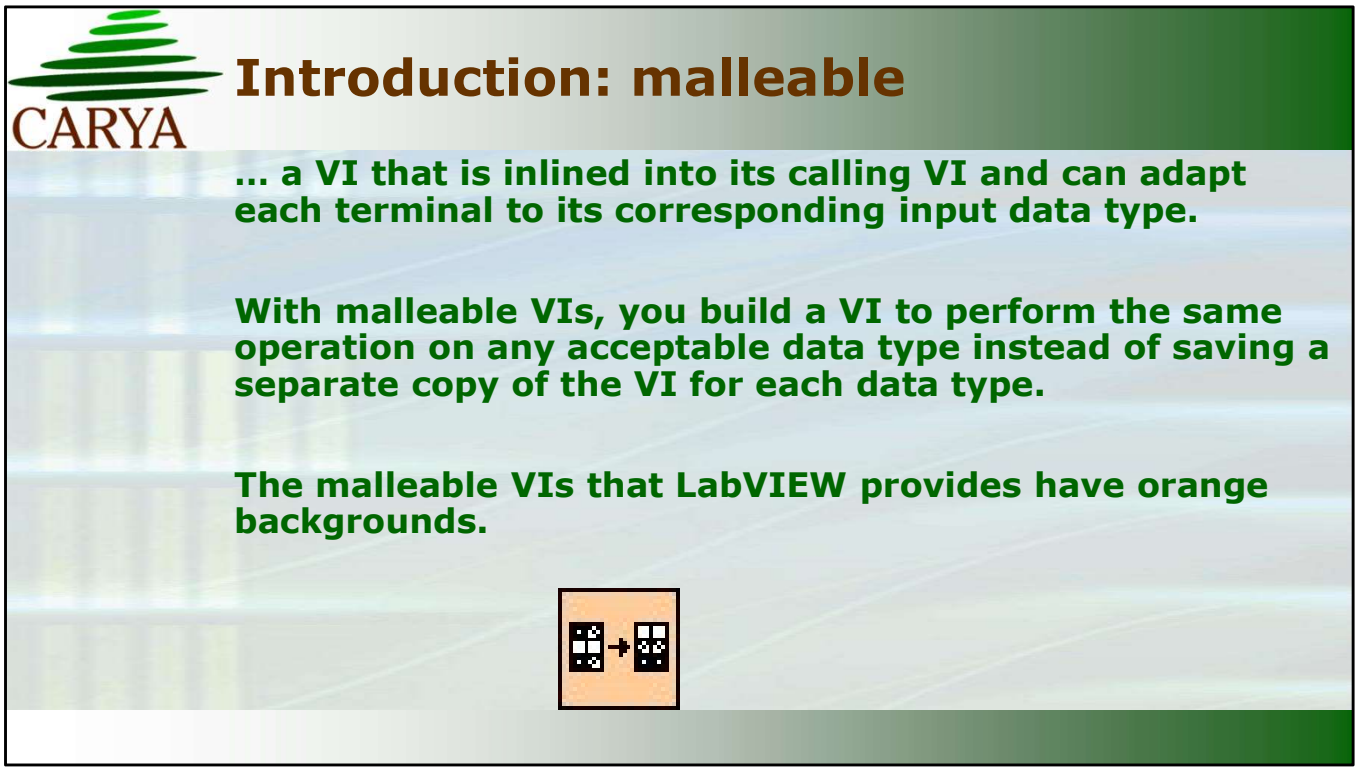
Synonyms

- **Impressionable, adaptable, mouldable, flexible, pliable, tractable**

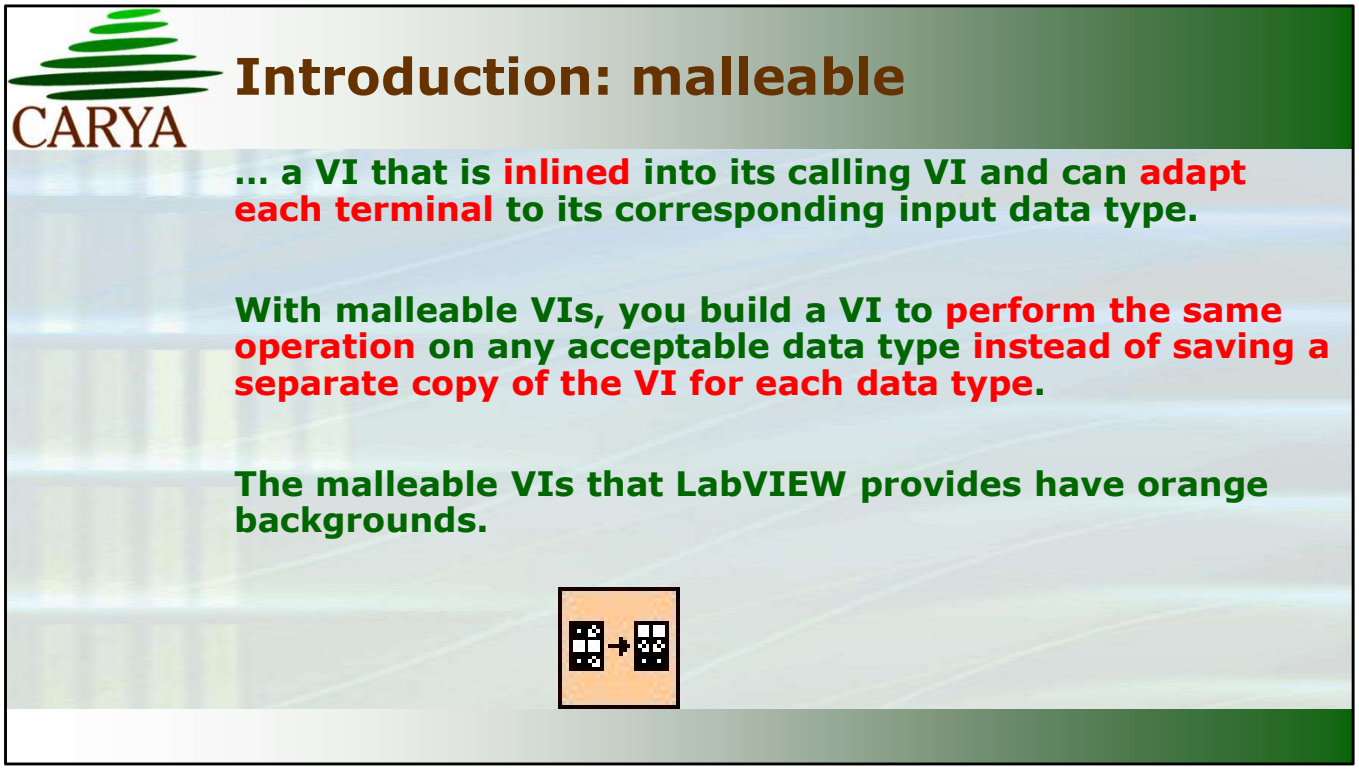
Origin

- **from Latin *malleus*: hammer**

That might already give some idea about what a malleable VIs is.



This is what the LabVIEW Help has to say.

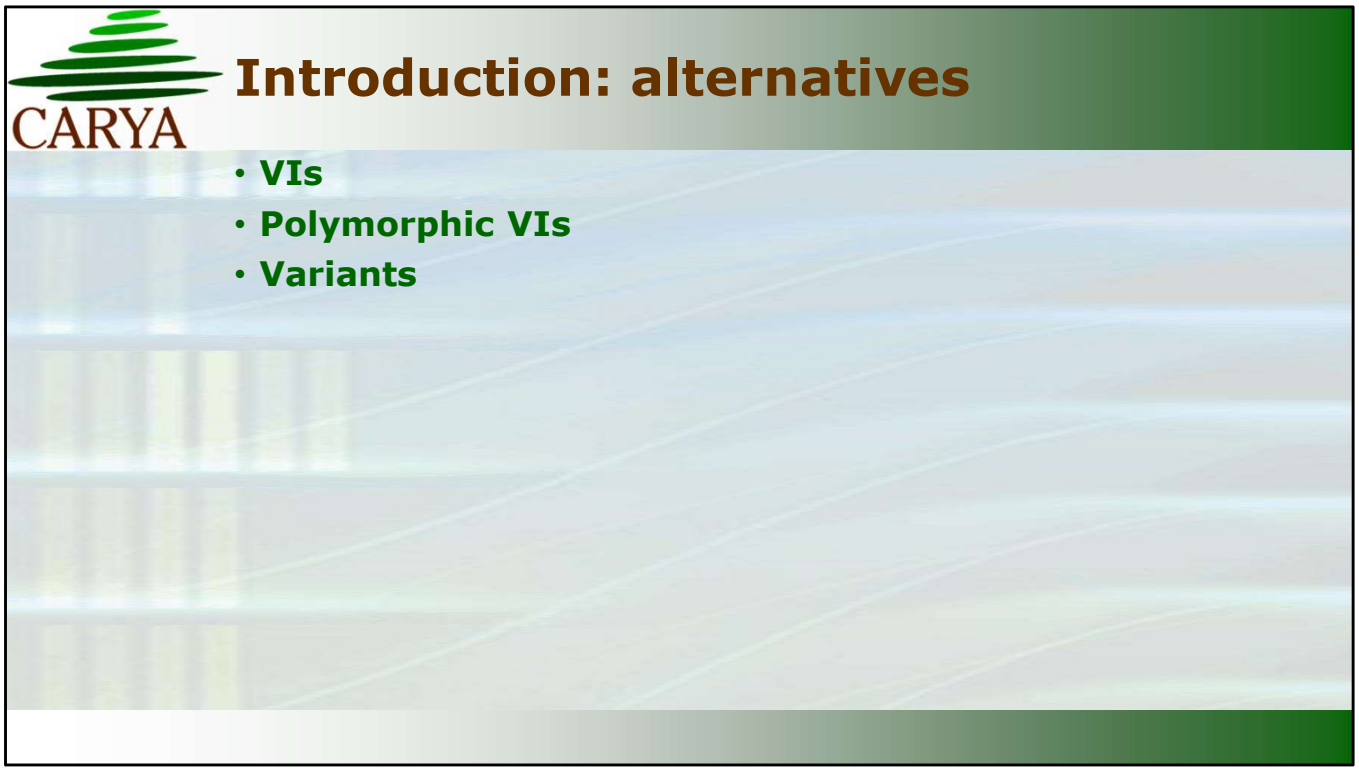


From the LabVIEW Help.

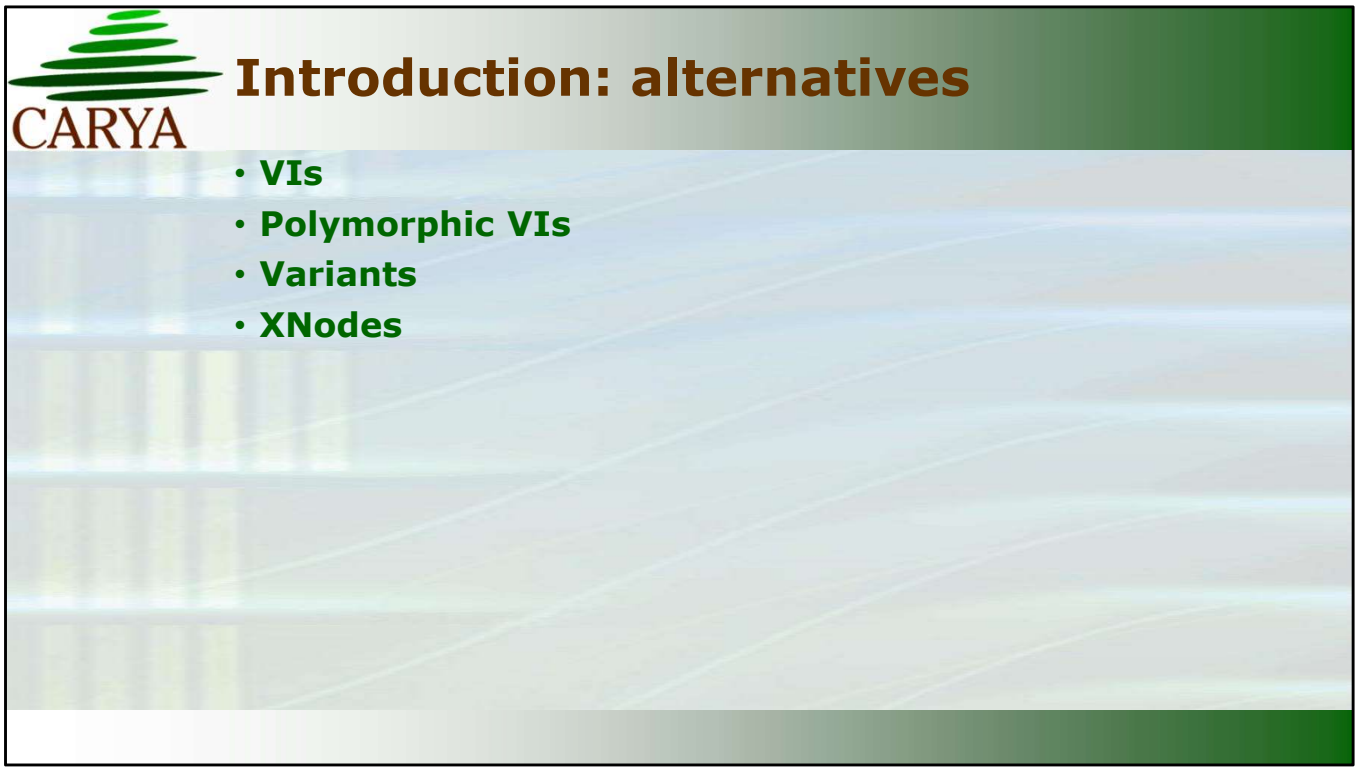
Lets take a step back and look at some of the alternatives, even though malleable VIs are not jet clearly explained.

VIs. Well nobody would like to have 20 VIs that do exactly the same! Not very user friendly.

Polymorphic VIs provide a nice wrapper for the user around these VIs, but I still need to make them. And maintain them.



Variants are a pretty good way to provide general purpose interfaces. But for instance doing math on variants is very difficult. Inside such a VI, you still need to convert back, using the variant type.

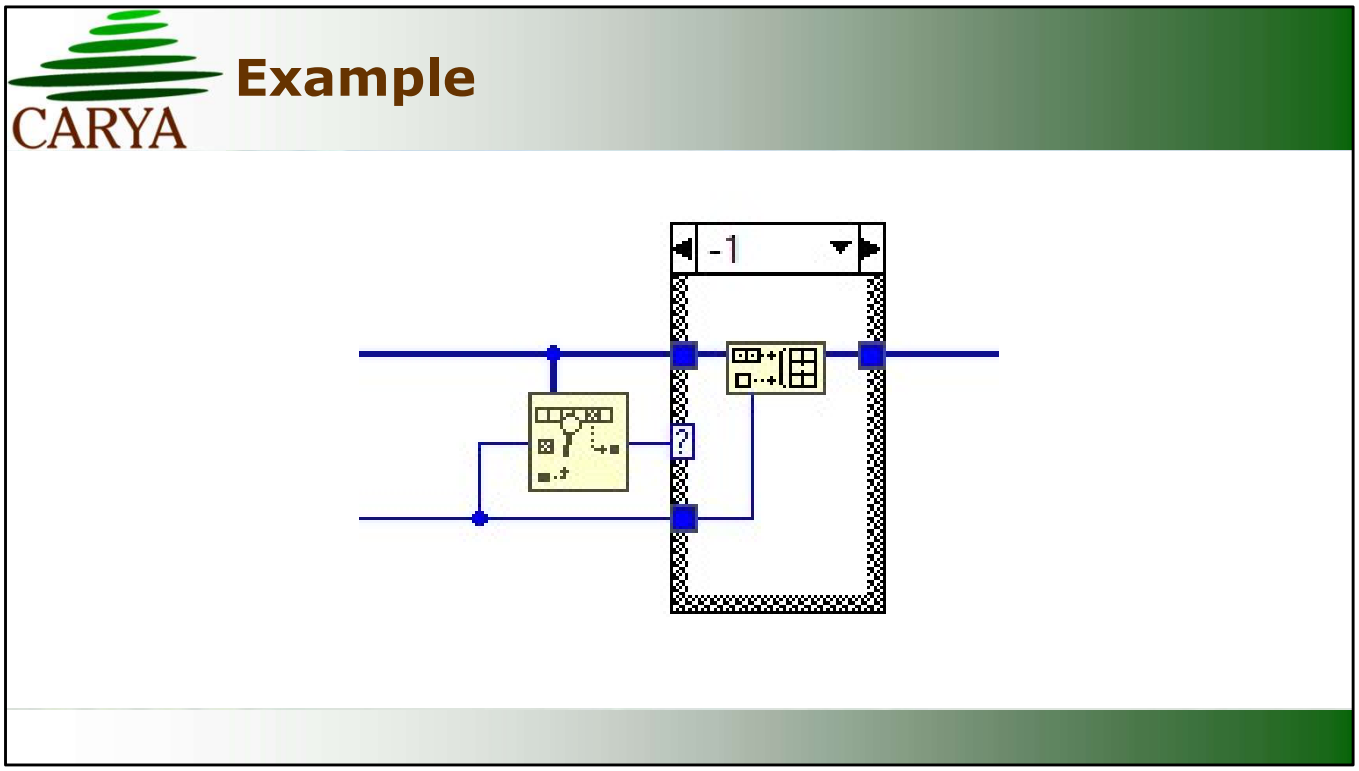


XNodes have never been officially released. They work very similar to XControls. LabVIEW calls VIs made by a programmer. Those VIs use scripting to create an inlined peace of code. You need a license for full IDE support, but there is some documentation and templates on LAVA. XNodes can't live in classes.

The first malleable Vis implementations used XNodes.

In all four alternatives, it's very difficult to support all data types. Clusters can contain arrays, arrays can contain clusters, and this can be repeated.


CARYA
Agenda
• Introduction
• Example
• Build in Malleable VIs
• New nodes
• Malleable VIs and OO
• Limitations
• Questions



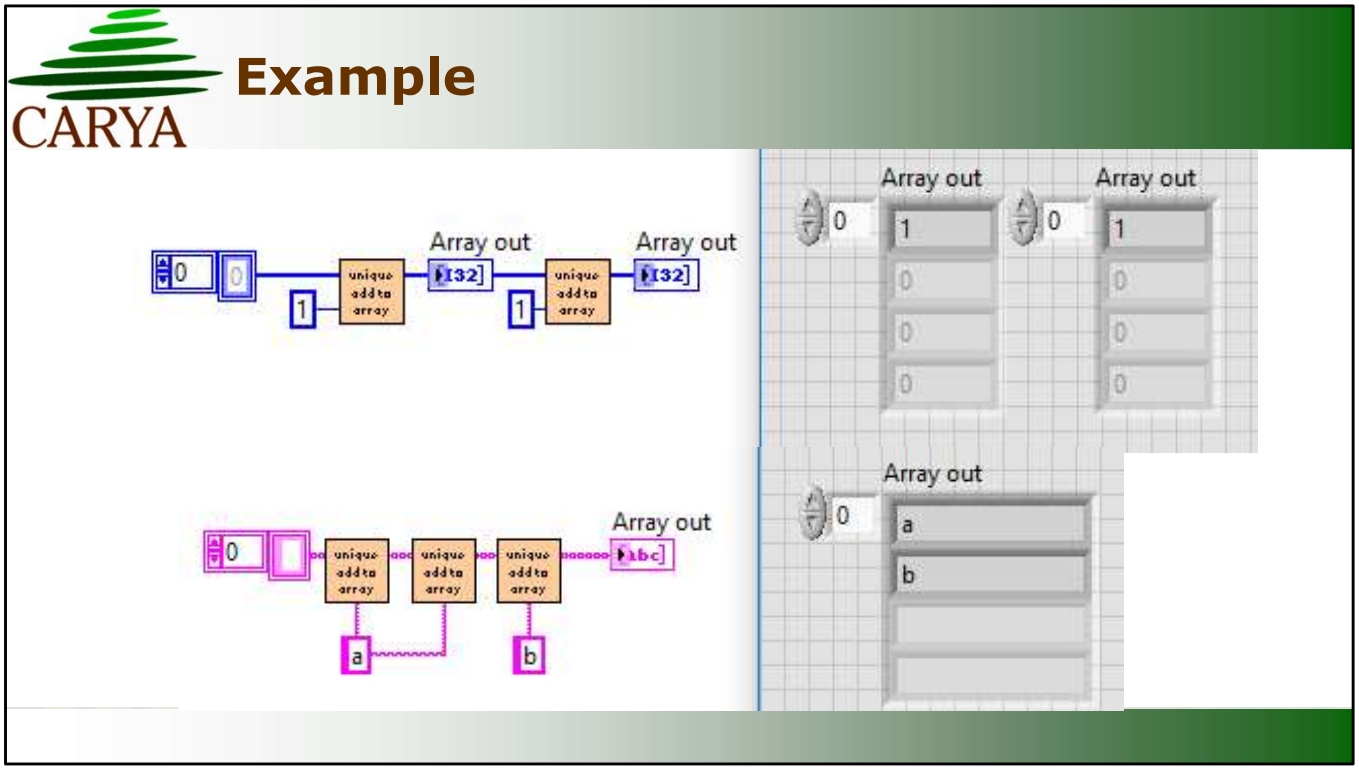
This is something I've made a lot of times. There's no point in making a sub VI! I still make one, but usually one ends up in every library, as each library has it's own data type.

Demonstrate the two ways to make a malleable VI.

1) New vim… Create, save
2) New vi, save as (don't forget it needs to be inlined!)

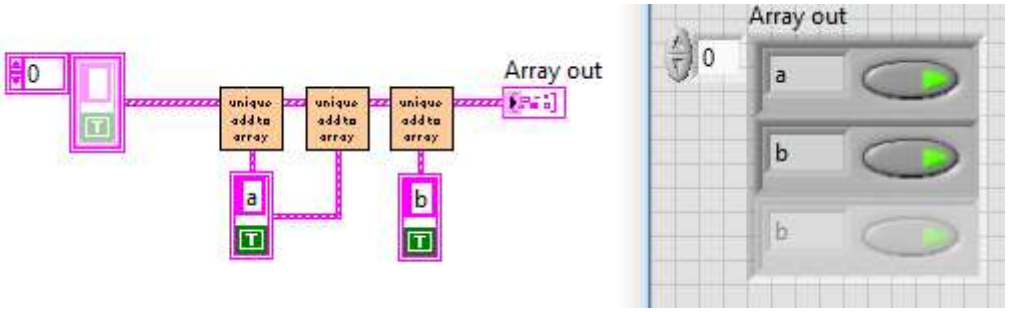
Demonstrate usage: integer, double, string.

Example

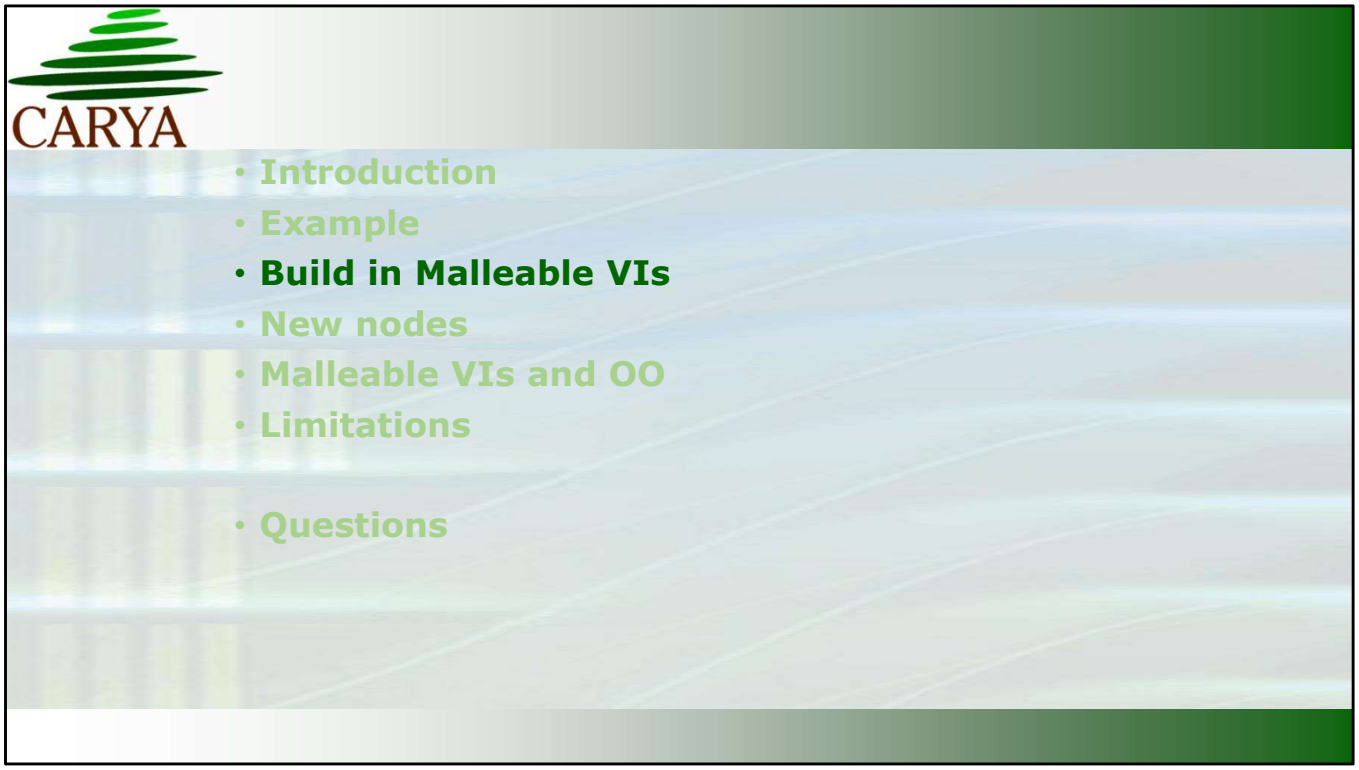


So all these sub VIs are the same peace of code!

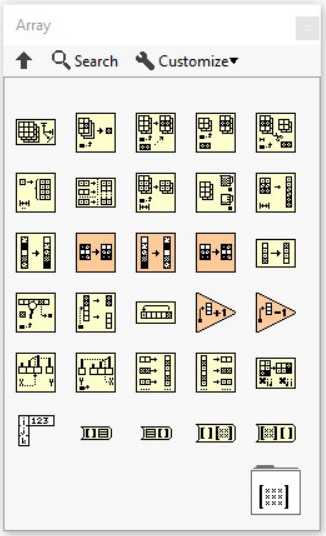
Example

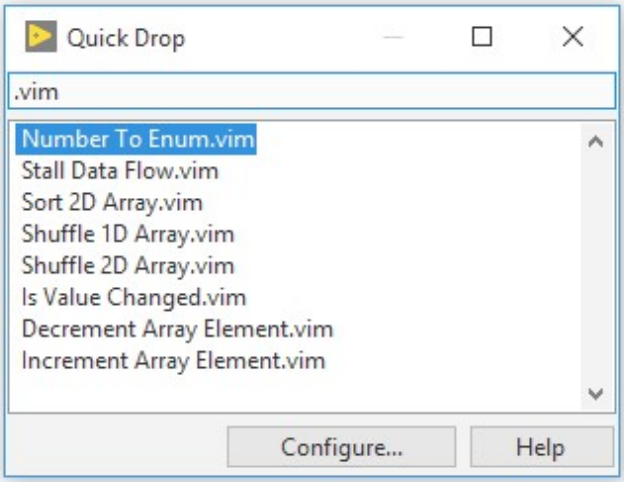


And it will even adapt to clusters.


CARYA
• Introduction
• Example
• Build in Malleable VIs
• New nodes
• Malleable VIs and OO
• Limitations
• Questions

Build in Malleable VIs

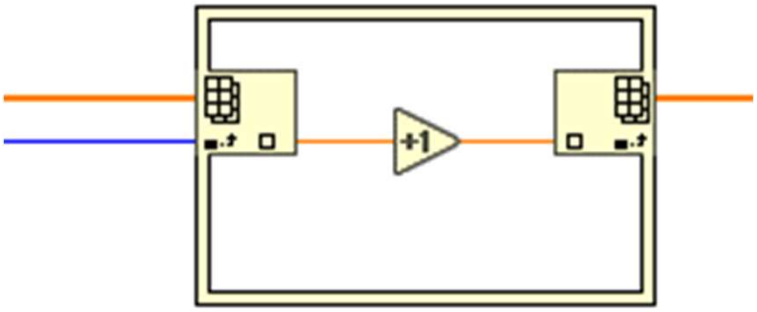




There are a few malleable VIs build in in LabVIEW.

Sort 2D Array, Shuffle 1D Array, Shuffle 2D Array, Increment 1D Array, Increment 2D Array.

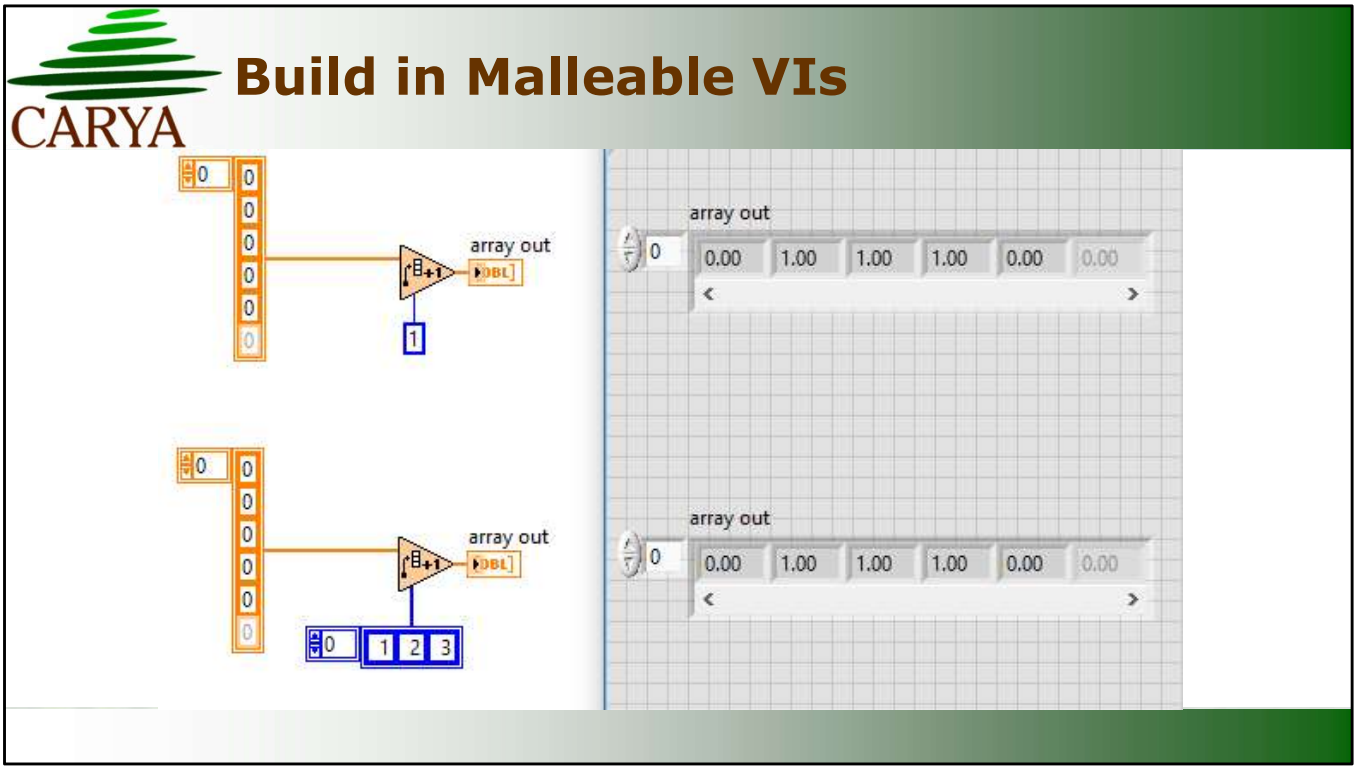
Build in Malleable VIs



Increment 1D Array does this in a malleable.

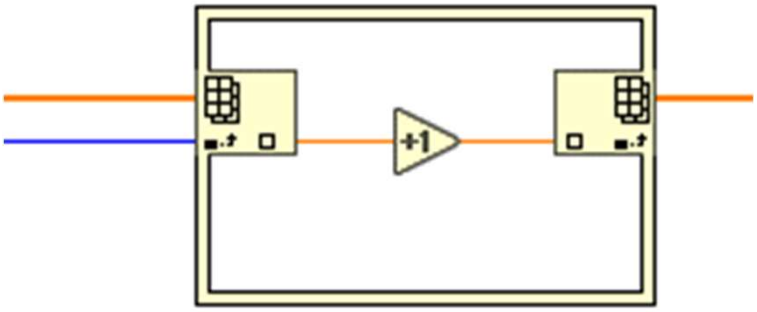
Demonstrate how increment array elements adapts to an array of indices.

Build in Malleable VIs



Note that it adapts to an array of indices!

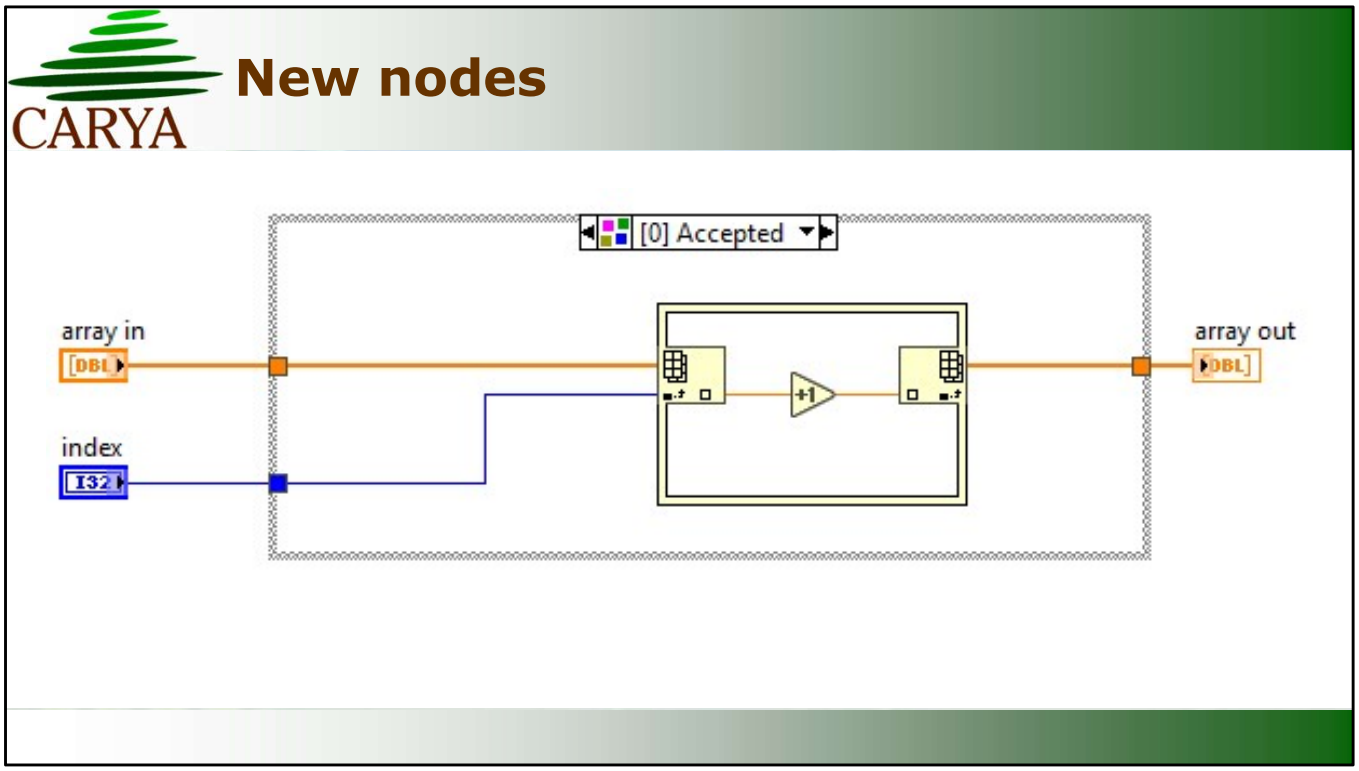
Build in Malleable VIs



How does that work?

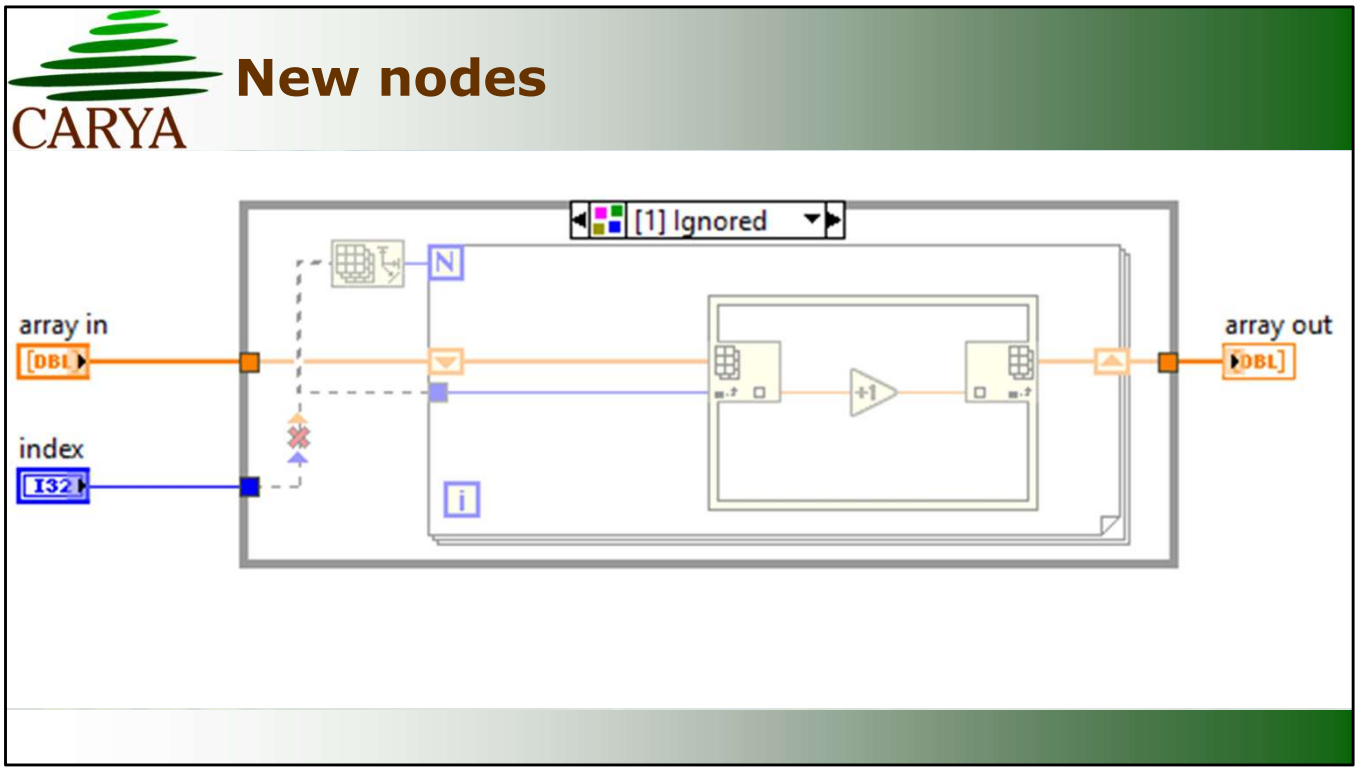
Surely this code won't accept an array of indices?

That brings us to the new nodes that are created just for malleable VIs

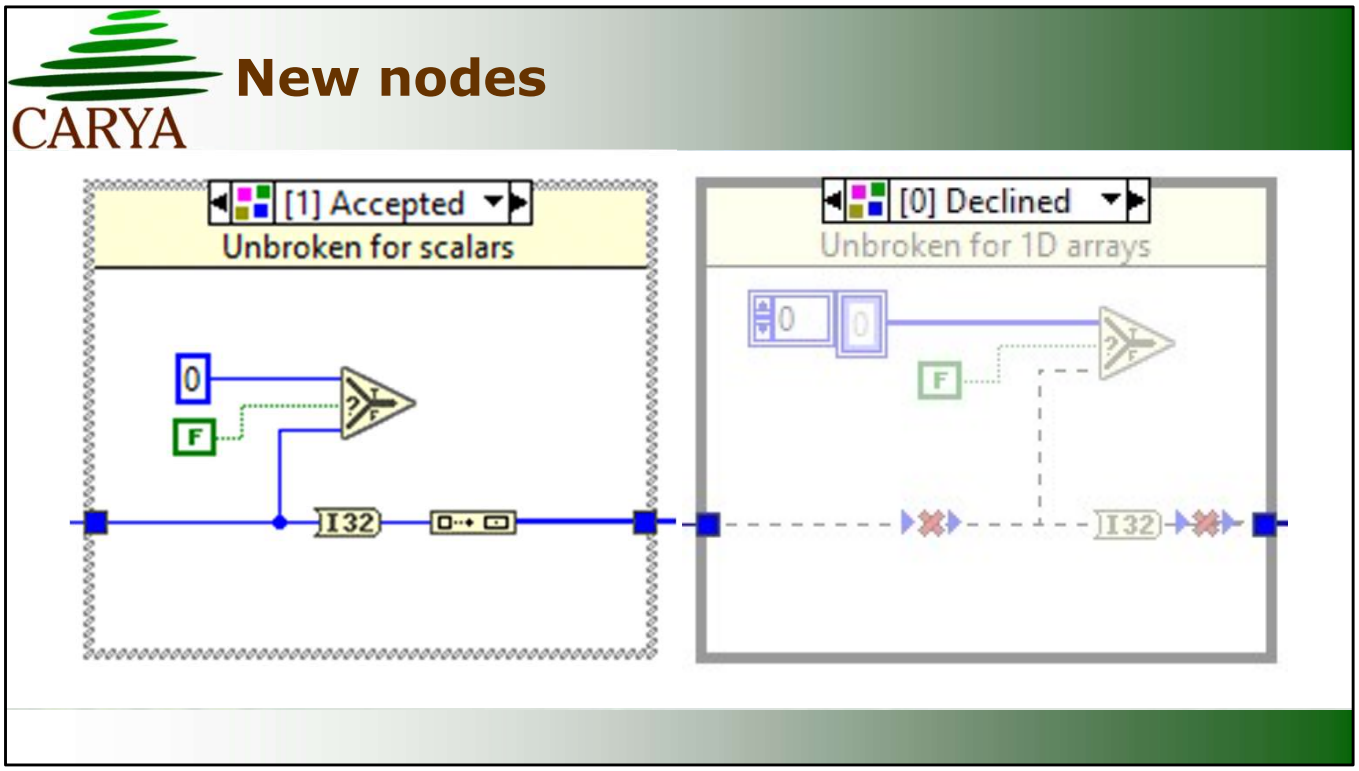


It can adapt to the array of indices, because it uses a Type Specialization Structure.

This structure is almost like a disabled structure. The difference is that you don't manually select the used case, but the compiler picks the first that's not broken.



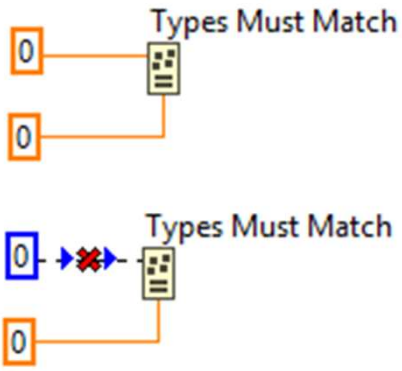
This case is disabled in the VI while it's being developed. But if an array is wired to index, this case is the first that's not broken. So it's used.



This requires some creativity…

But both of these still cannot distinguish between double and I32. The select would simply adapt.

New nodes

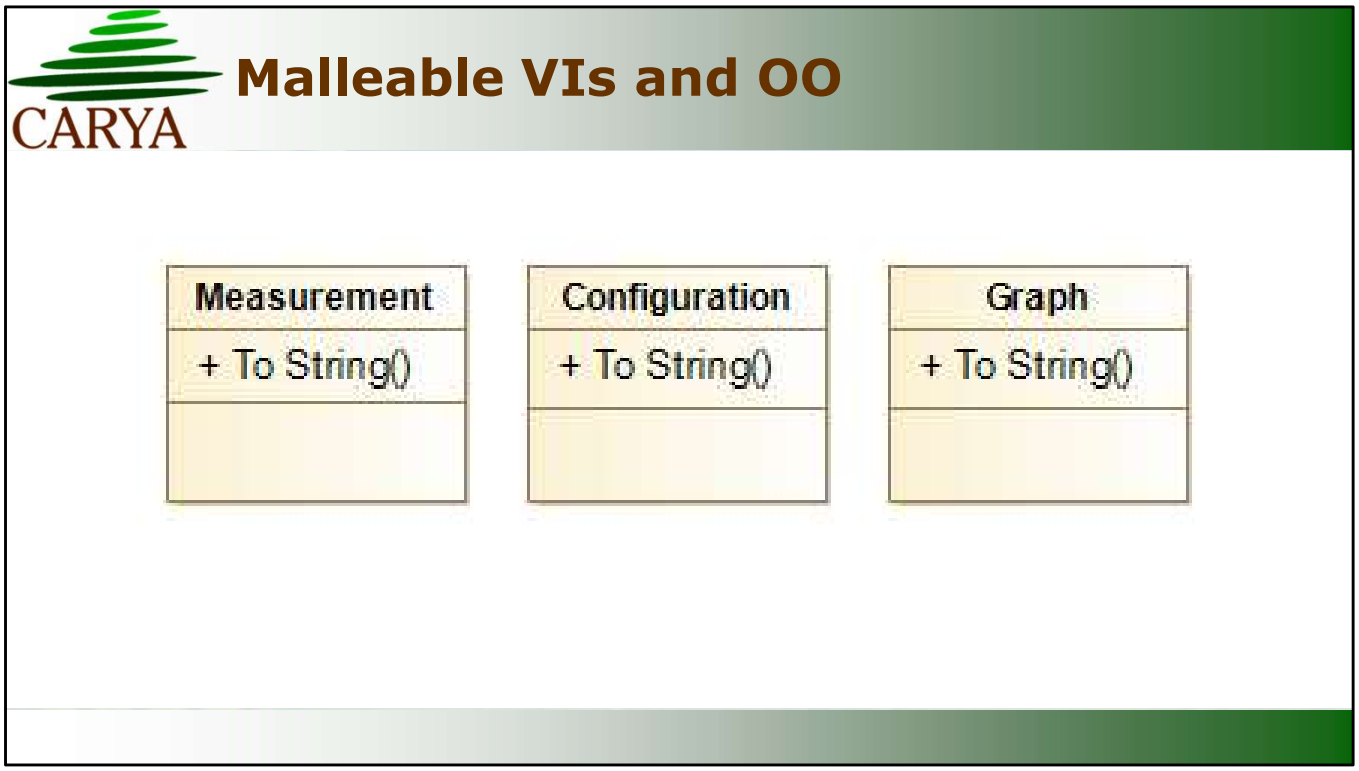


One function that helps is the new Types Must Match function.

It ignores names (e.g. in clusters and arrays), but the types must match exactly. For instance an add will accept an array and a scalar, a cluster and a scalar, and a double and an U8 without any problems

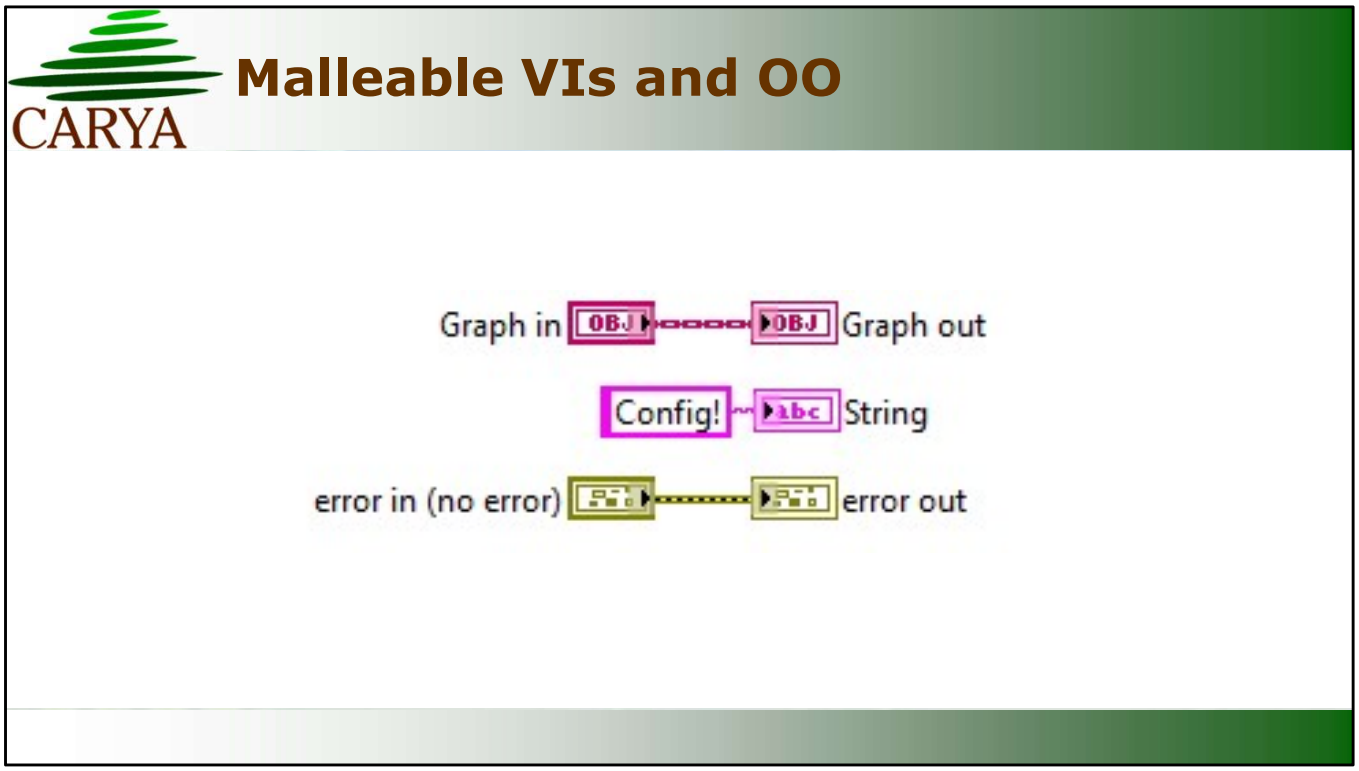
You'd most often use Types Must Match in a Type Specialization Structure, to help it select the right case.


CARYA
• Introduction
• Example
• Build in Malleable VIs
• New nodes
• Malleable VIs and OO
• Limitations
• Questions



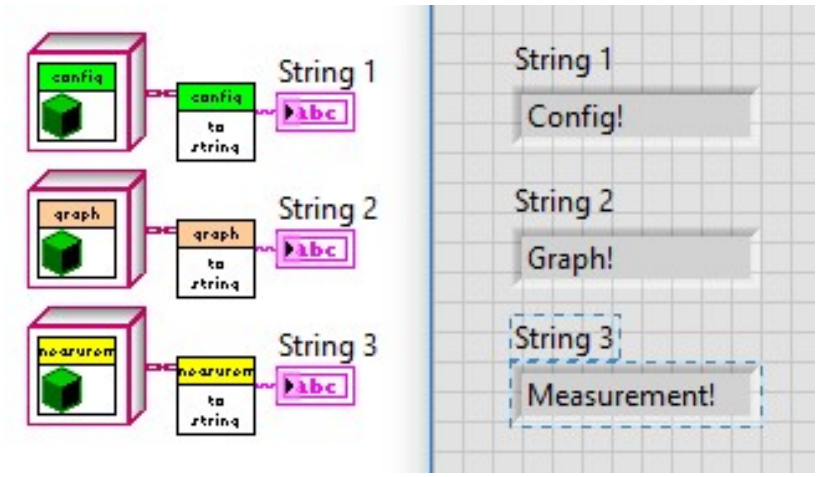
Let's say we have three classes. Each one has (among other methods, not shown here) a To String method.

We can use those methods everywhere.



This is what's in them. This is the graph, it simply outputs a constant.

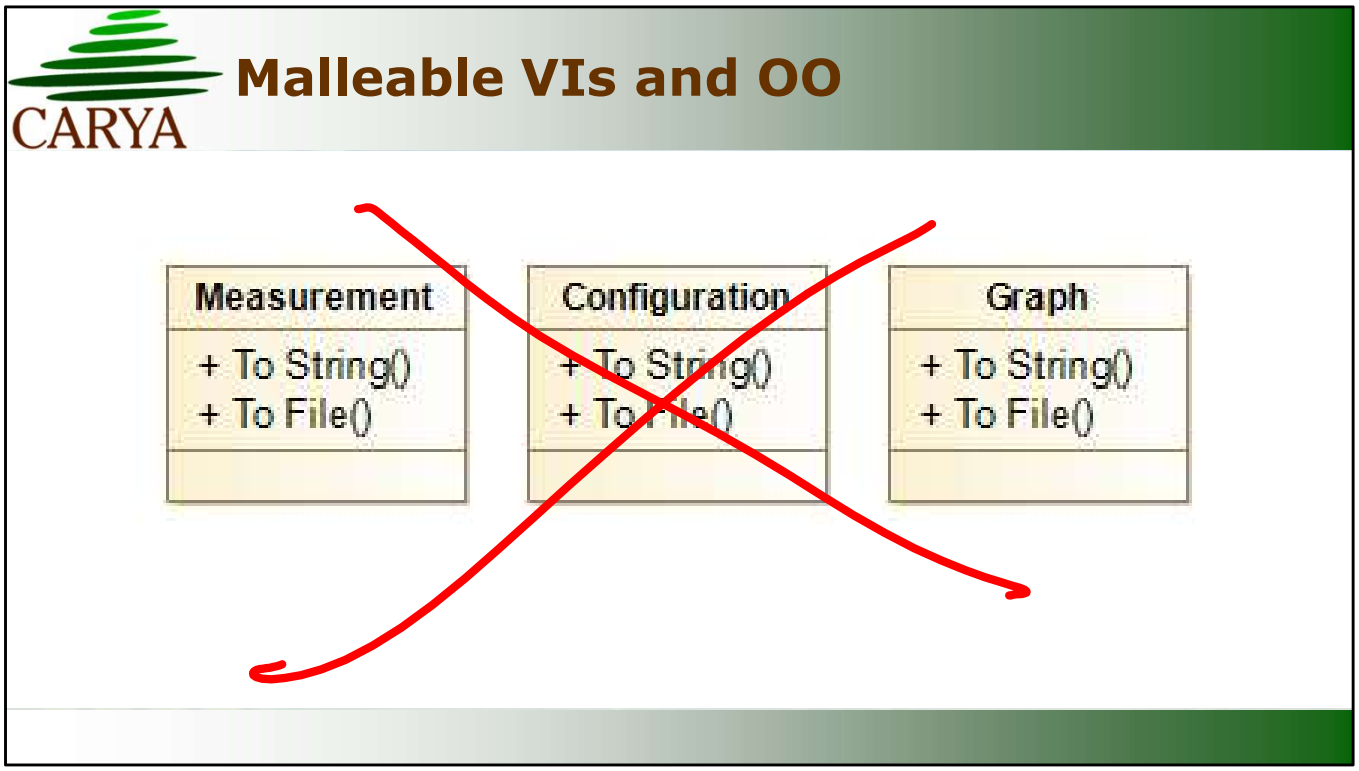
Malleable VIs and OO



Here's how we can use them.

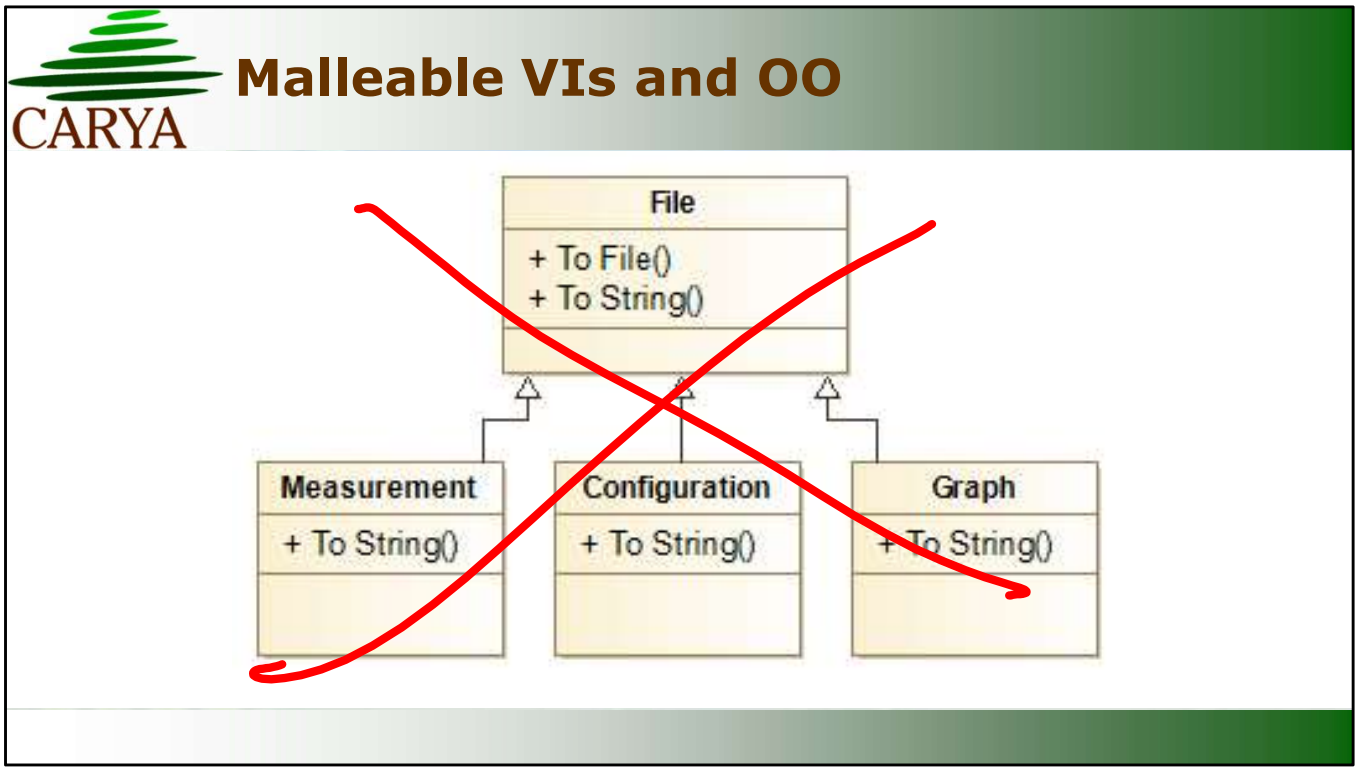
But we **cannot** put them in an array and call To String. The parent (a LabVIEW object) does not have a To String method.

However, it also prevent us from making a To File **VI**. Or a To Database **VI**.



One way to implement them is to add them to each class.

Those methods would look exactly the same for each class.



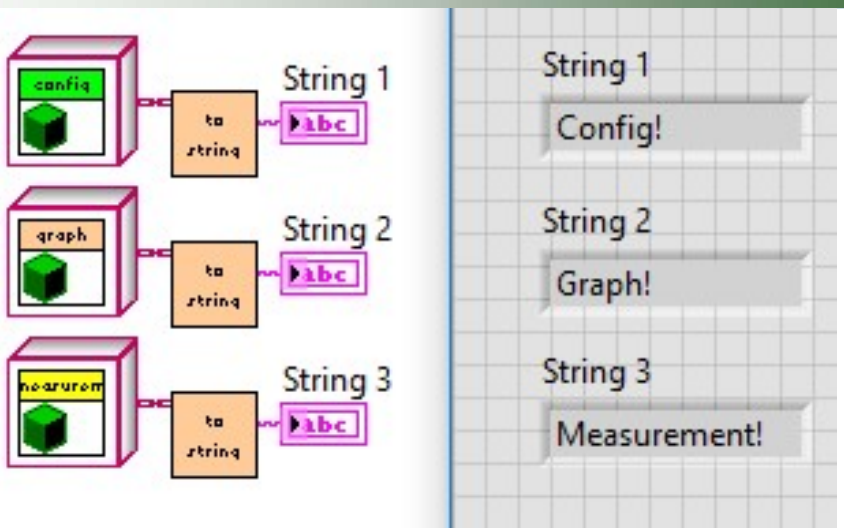
Or we can give those classes the same parent.

To File can dynamic dispatch the To String method.

This does not make sense at all, because a measurement is not a file. And a graph even less so.

But what if we want to add a To Database method? That's not possible.

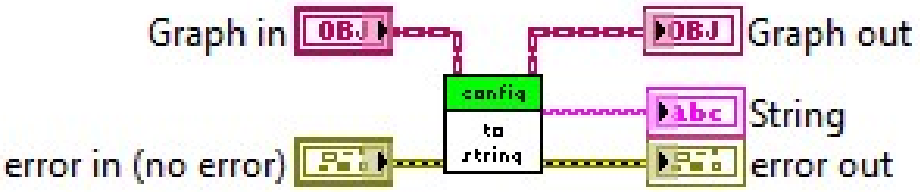
Malleable VIs and OO



Malleable VIs will simply replace the method we used in the vim with a method with the same name.

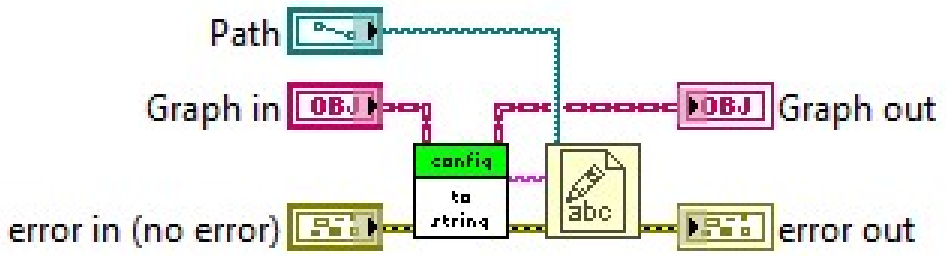
This is just one VI.

Malleable VIs and OO

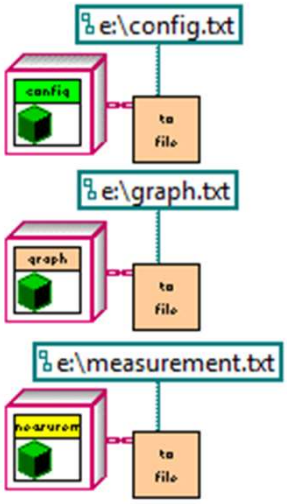


And this is how it would look.

Malleable VIs and OO



Malleable VIs and OO



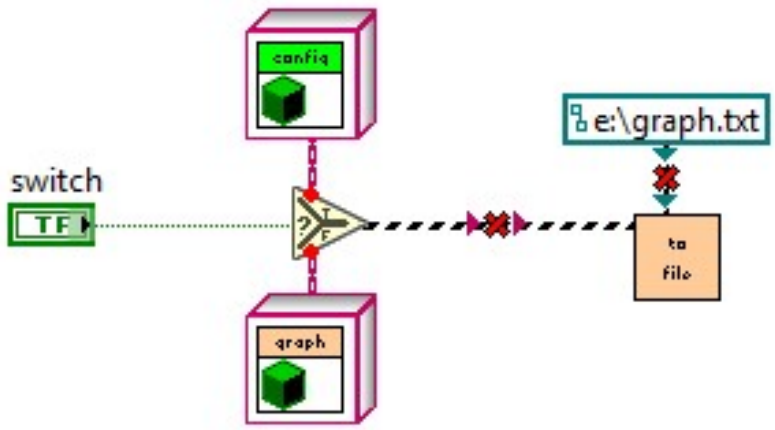
This is again all the same VI.

Note that we used the To String VI from those classes.

We don't need all of them to return a string. For instance one class might return an integer, the other a string, the third a double. Let's call it Get Data.

The calling malleable can use format into string, and that will fit all of the VIs! This means even less code to make.

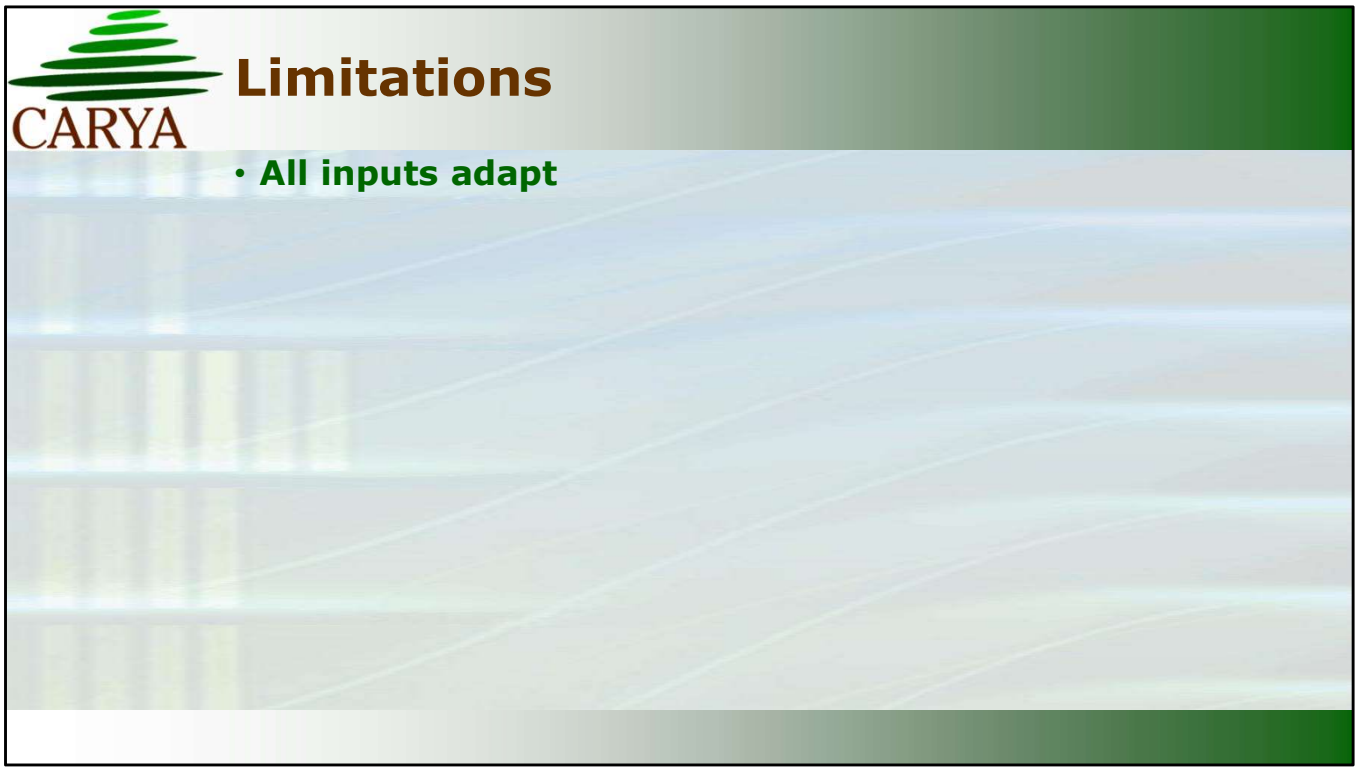
Malleable VIs and OO



This still won't work. The vim needs to be resolved at compile time.

To do this we need Dynamic Dispatch, with all the downsides. Or Interfaces. But Interfaces will take a while (maybe LV2019 or LV2020).


CARYA
• Introduction
• Example
• Build in Malleable VIs
• New nodes
• Malleable VIs and OO
• Limitations
• Questions



As all inputs adapt, you lose the option to get a variant. In a normal VI, the input can be a variant, and that variant will have a label. With .vim's, that's not possible.

This also means the feedback to the user is not always that clear. You can wire anything, and there are two options. The wires will adapt, or the wires will break. But when one input is an array of doubles, inserting an integer is accepted, but there will be no coercion dot.

Note that polymorphic VIs are still useful in some usecases. One big difference is that Polym. VIs can have a different number of CP items.

.vim's must be inlined (and thus reentrant).

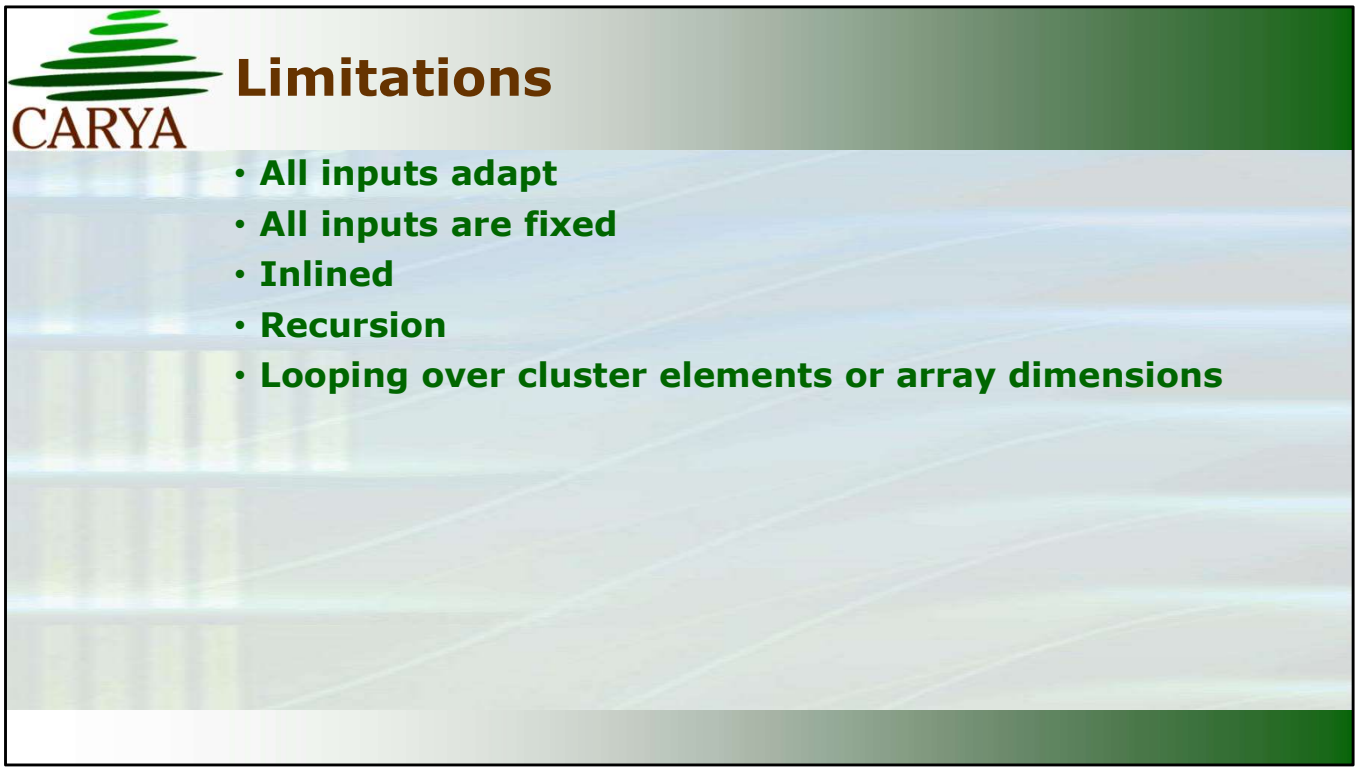
This means property and invoke nodes are not excepted. Also expression nodes are not allowed.

A vim smart buffer is not possible.

.vim's do not allow recursive calls (cannot call itself). This is actually true for all inlined VIs.

It makes sense. If the VI is inlined, and contains itself it's inlined, but that one also contains itself. Etc. until the compiler crashes.

But for .vim's it's very inconvenient. For instance, a To String function is easy for all scalars. But when a cluster is wired, it makes sense that the To String function is called for each element.



First of all, looping over cluster elements is not possible because reentrancy is not allowed. But looping over each cluster element is also not possible, because there is no function for that.

With arrays there is a similar problem. For instance a 1D search function, could be extended for 2D, 3D, nD. But the nD would call the same VI recursively in a for loop, which is not allowed.

Only solution for now is to flatten the array (memory copy), or to manually implement it for 1D-10D arrays.


CARYA
Questions