

Additional Notes on Kulula

Matthias Sala

`salam@student.ethz.ch`

Department of Computer Science

Programming Languages and Runtime Systems Research Group

Swiss Federal Institute of Technology (ETH) Zurich, Switzerland

November 23, 2005

Additional Notes on Kulula

Introduction. Kulula is a language to describe the skeleton of a new component. It contains component composition details (declarations), visual layout constraints (**SEE** section), component wiring information (**ACT** section) and description of data (**DATA** section). Additionally, one can define wiring interfaces open to other components (**OFFER** section).

You cannot build complete applications with Kulula. It is more part of a full development cycle [1]:

“Firstly, compose the different components meaningfully. Secondly, supplement the details in the emitted skeleton with a general purpose language”

Further explanation of principles in Kulula

The *OFFER* section defines wiring interfaces open to other components. Like that other components can access the new component with the given *role name*. A role name is a simple word that can be used in the **ACT** section of other components. E.g. the (hidden) text model component can offer the ability to be *displayed* (just the relevant lines of the code are shown - but there is more between the lines):

```
THE Text
```

```
OFFER
```

```
  displays
```

```
    OTHERS SetText WITH MY GetContent AS String ON MY Change
```

```
    MY SetContent WITH OTHERS GetText AS String ON OTHERS Change.
```

A component that uses the text model is the *Editor*. The editor has a visual text field that displays the abstract hidden text model (just the relevant lines of the code are shown - but there is more between the lines):

```
THE Texteditor
```

```
  model AS Text
```

```
SEE
```

```
  view AS Textfield AT CENTER WITH MULTILINE
```

```
ACT
```

```
  view displays model.
```

Implementation details not mentioned in the student project report

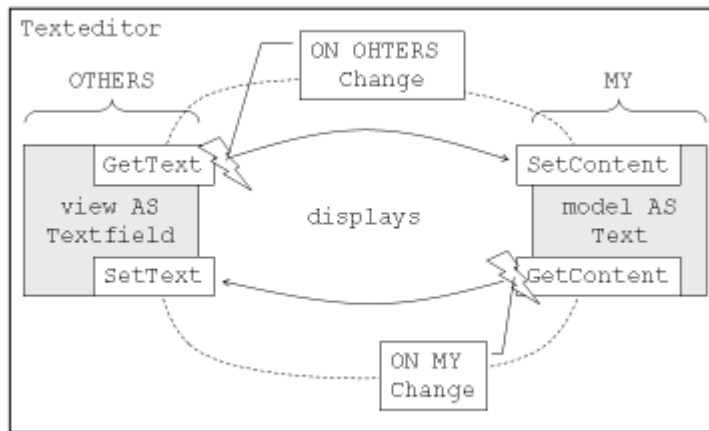


Figure 1: The role *displays* between *view* and *model*

Foundation classes are root components that build the bridge between the output language and the skeletons builded. A foundation class consists of

- a .Kulula file, *Text.Kulula* (to indicate the compiler to stop compile, it extends from a terminator: `The Text OF KululaFoundationClass`),
- a runtime binary, *KFCText.Mod / KFCText.Bin* (in our case a MODULE binary that implements the procedures necessary), and
- a part in the configuration file (`Kulula.XML \ Tag Types \ Tag Type` with attribute `warp="Text"` and an attribute `oberon="KFCText.Text"`)

Recursive compiling is not done by the Kulula compiler. E.g. when compiling the sample *SubContainer* from E.4 in [1], the subcomponents are not compiled automatically. Since the subcomponents may have been supplemented manually, an automatic compilation might overwrite the supplemented skeleton.

Nevertheless, the compiler inspects the IR of the subcomponents for semantics. For example, the compilation of *Editor.Kulula* works as follows:

1. Read in the .Kulula source file
2. Build the IR
3. Read the types involved in the hidden and visual declarations ("`model AS Text`" → `Kulula.XML \ Tag Types \ Tag Type` with attribute `warp="Text"` → attribute `oberon=?` → build IR of *KFC-Text.Kulula* recursively → it's a foundation class!)

4. Open a .Mod output file
5. Iterate IR and emit code (check `Kulula.XML \ Tag property` for that reason)
6. Implement necessary handler methods (wiring of components on events call handler procedures)
7. Append positioning functions used (`Kulula.XML \ Tag Positioning \ Tag Function` with attribute `name`)
8. Store file
9. Compile it with the Oberon Bluebottle parallel compiler PC0
10. Print `Editor.Start` to log console

Limits of the current implementation

The text model component is a foundation class (therefore the OF `KululaFoundationClass` inheritance). In the current state, it uses a buggy reader functionality that seems to stop at some control characters (the affected code line in `KFCText.Mod` is marked with a all capital letter comment). And it uses a big array to build up a record of type *String*. One might use other dynamic array allocation methods to be more memory economical, or to increase speed (see `CONST MAXSIZE = 24000`).

The implementation of *MediaPanel* found in `MediaPanel.Mod` is incomplete. The rendering is propagated not at all or to the wrong place on the virtual screen. The procedure `ShowFrame(frame: Raster.Image)` in type `MediaPanel` in module `MediaPanel.Mod` has to be fixed.

XML tags can be assigned in the init procedure, since all Kulula components are Bluebottle `XMLElements`, but the data section is not imported, yet.

Additional configurations in the next Bluebottle release

The *Decoders* section in `AosConfig.XML` might be enhanced with

```
<setting name="Kulula" value="AosTextUtilities.UFT8DecoderFactory" />
```

to offer a choice for format in the Editor and PET.

The *Filehandlers* section in `AosConfig.XML` might be enhanced with

```
<setting name="kulula" value="Notepad.OpenAuto" />
```

to link the Kulula files with the Bluebottle Notepad.

Proposals

The **BUT NOT clause** is an interesting additional feature to macerate a wiring contract. The idents after the BUT NOT clause are subtractive to the the difference between the wiring interfaces of the left (L) and right (R) side of the contract (L rolename R). The semantic checker might supress a warning or alert, and the code emitted might produce an empty handler as a slacky end of a wire.

In the property list of a hidden or visible subcomponent: the strings and numbers might be moved to an XML tag, as well.

Iterators and arrays for declarations are an interesting way to create toolbar buttons faster. Unfortunately, it is less ergonomic for the user since the names have anonymous numbers in it. An iterative declaration might look something like:

SEE

10 TIMES

```
apanel AS Panel AT TOP WITH BORDER, "apanel with index THIS apanel"
```

AND

5 TIMES

```
abutton AS Button AT LEFT IN THIS apanel WITH "abutton with index THIS abutton  
in apanel with index THIS apanel"
```

...

ACT

5. abutton uses 7. abutton

where 10 TIMES starts a loop and appends a numerical index to the ident name `abutton` (EBNF: *Loop* \equiv *NumberOfRepetitions TIMES Declaration {AND Declaration}*.). The keyword *THIS* replaces itself and the following ident with the actual number of it in the loop (EBNF: *ArrayIndex* \equiv *THIS ArrayIdent*.). The number-ident pair indicates an item of an array. For the compiler, the dot after the number indicates that it is an index of an array (EBNF: *ArrayItem* \equiv *IndexNumber "." ArrayIdent*.).

Sources & Binaries. Please take the newest versions of Kulula from [2].

apanel with index 0	abutton with index 0 in apanel with index 0	...	abutton with index 5 in apanel with index 0
apanel with index 1	abutton with index 0 in apanel with index 1	...	abutton with index 5 in apanel with index 1
...			
apanel with index 10	abutton with index 0 in apanel with index 10	...	abutton with index 5 in apanel with index 10

Figure 2: Visual result of iterator example

References

- [1] M. Sala, Kulula - a Text-based Domain Specific Programming Language To Wire Components, *Student Project*, 2005.
- [2] Kulula website, www.n.ethz.ch/student/salam/kulula.