



Kulula -
A Text-based Domain Specific Programming
Language To Wire Components

Matthias Christoph Sala
salam@student.ethz.ch

Student Project¹
May 2005

¹This is the report of the student project written during a study abroad semester at the Department of Computer Science, University of Stellenbosch, South Africa. The supervisor of the student project was Prof. Jürg Gutknecht (gutknecht@inf.ethz.ch), Programming Languages and Runtime Systems Research Group, Computer Systems Institute, Department of Computer Science, Federal Institute of Technology, Zurich. The technical support person for the *Bluebottle* system was Thomas Frey (frey@inf.ethz.ch). The contact person in Stellenbosch was Prof. Pieter de Villiers (pja@cs.sun.ac.za).

Abstract

In the past, many different attempts for composing components, especially multimedia components, in a visual manner have been proposed. However, we believe that a classical text-based programming language can be very useful for specific tasks. In this paper we introduce a new language that enables to create nearly complete application in component-based environments like the *Active Oberon's Bluebottle* system. This language is named Kulula, the expression for '*It is simple*' in *Xhosa*, a South African language.

Contents

1	Introduction	4
2	Language Design	6
2.1	Syntax Simplification	6
2.2	Component Technology	6
2.3	Hiding Complexity And Optional Properties	6
2.4	Layout	7
2.4.1	Text-based WYSIWYG	7
2.4.2	Positioning Functions	7
2.4.3	Containers	8
3	Abstract Wiring	8
3.1	Composing and Supplement	8
3.2	Event-Handler Pattern	8
3.3	Roles	9
3.3.1	Connect With Wiring Contracts	9
3.3.2	Role Definition	11
3.3.3	Naming Convention	11
4	Dynamic Data	12
4.1	Data Isolation	12
4.2	Data Import	12
4.3	Internationalisation	13
5	Expandability	13
5.1	Inheritance	14
5.2	Configuration	15
5.3	Platform And Language Independence	17
6	Conclusion	18
6.1	Problems	18
6.1.1	Data Redundancy And Conflicts	18
6.1.2	Wiring And Parameters	18
6.2	Possible Improvements	19
6.2.1	Semantic Checks And <i>BUT NOT</i> Clause	19
6.2.2	Foundation Components	19
6.2.3	More Language Independence	19

6.2.4	Automatic Data Extraction	19
6.2.5	Cross-Platform Compiler	20
6.2.6	Generated Scanner And Parser	20
6.2.7	Extended Property And Data Types	20
6.2.8	Integration Into A General Purpose Language	20
A	Compiler Design	21
A.1	Scanning And Parsing	21
A.2	Semantic Checks	21
A.3	Code Generation	22
A.4	Runtime Services	23
A.4.1	XML Queries	23
A.4.2	Component Monitor And Control	23
A.5	Modules Implemented	23
B	Component Skeleton	24
B.1	Hierarchy	25
B.2	Declarations	25
B.3	Initialisation	25
B.4	Event-Handlers	25
B.5	Positioning Functions	25
C	Runtime Environment	26
C.1	Instantiation	26
C.1.1	Loading	26
C.1.2	Wiring	26
C.2	Event-Handler Relation	27
C.2.1	Normal Scenario	27
C.2.2	Loop-Protection And Locking	27
C.3	Dynamic Data Retrieval	28
C.3.1	Reflection	28
C.4	Termination	29
C.4.1	Normal Termination	29
C.4.2	Forced Termination	29
D	Step-by-step Example: A Simple Text Editor	29
D.1	Compose	29
D.2	Generate skeleton	30

D.3	Run	31
D.4	Supplement	32
D.5	Compile	32
D.6	Run	32
E	More Examples	32
E.1	Control Nightmare introducing Placement Functions	32
E.2	FilenamePicker and BetterFilenamePicker introducing Inheritance	34
E.3	Mediaplayer in 40 Words Only	35
E.4	SubContainer, et al. introducing Reuse	36
F	Language Specification In BNF	38

1 Introduction

Since the beginning of Computer Science, the scientists were trying to make development easier for industry members. They want to use the technology as a tool, not as a science. Therefore, many attempts have been made so far: object-oriented and aspect-oriented programming, scripting languages, visual editors, component-based frameworks, or even artificially learning programs.

Unfortunately, none of the proposed solutions are widely used nor accepted until today. Even easy applications with databases need CS experts, very often. A possible approach is the old idea of domain specific programming languages (DSL) where users that are experts in their domain (e.g. biologists, sociologists, marketing specialists) can develop applications without a deep understanding of algorithms and data structures.

For the area of presentations and multimedia applications a lot of useful products have been published (e.g. by *Macromedia Flash*) where the user can program in a visual manner with real-world semantics (layers, time lines). All these products reach a limit where this semantics are not powerful enough anymore. It is where they introduce script languages (*ActionScript*[9]). A big mistake is that c-sibling general purpose programming languages are chosen, most of the time. Anyway, a DSL is dedicated to fill the gap between the needs and the lack of the visual semantics.

Further, a problem of existing component-based frameworks is their tight mapping to underlying software principles. It would be interesting to develop a different kind of wiring between components. The *Open Services Gateway initiative*[7] uses Java listeners, and interfaces to communicate between services. The approach is very dynamic, but expensive for this reason. *Beanome*[8] addresses this problem and introduces a new model concept for OSGi. Additionally, Trygve Reenskaug has proposed some interesting theoretical aspects of wiring[5]. With this work we introduce a similar wiring strategy with roles.

Today, internationalization and customization are important factors in software development. Anyway, the newest programming languages like *Sun Java* and *Microsoft .Net* and *C#* do not offer language elements to manage these tasks. With Kulula all data will be externalized and loaded dynamically at runtime.

The goal of this work is to design a language and implement the corresponding compiler in Bluebottle[1]. The language is a DSL that emits a skeleton written in Oberon[4] to give the full power of a general purpose language.

The users of the language are programmers that want to program more efficiently saving time with standard tasks. Further, the language should have the following properties: easy to learn, ergonomic, hiding complexity, binding dynamic data and expandable.

2 Language Design

A language should be learnable in a short time. Therefore, a few clear and strong principles and rules should hold. First, we introduced a new abstraction level: applications consists of components, relations, new services and data. And we tried to loosen the traditional strict syntax rules and to reduce the visible complexity.

2.1 Syntax Simplification

Although the language has a strong syntax that can be expressed in BNF, the syntax rules seem to be more user-friendly than for traditional languages. There are no separators between statements, definitions and declarations. But line breaks with indents can help increasing legibility. Further, there are no *END* clauses in Kulula. Since the language does not offer nested definitions, the interpretation of any program text is never ambiguous. Finally, the number of keywords is little and no special characters are used.

2.2 Component Technology

Every Kulula program consists of a combination of components and represents a new component itself. Like that, we can use existing components, connect them, specify their properties and define new available services.

2.3 Hiding Complexity And Optional Properties

Components are units that fulfill a specific function, but one can adjust their functionality slightly via properties. Indicating such properties in other languages is very time-consuming. Normally, we want default values where just one parameter is different. Therefore, we have introduced optional properties in Kulula. When declaring an instance of a component, we can specify some properties inline. Here is an example:

SEE

```
text AS Textfield WITH "A sample text", MULTILINE, 2
```

The given property list is handled by the compiler using the configuration that specifies the meaning of the properties regarding their type (string, keyword or numeric) and their order. The string "*A sample text*" would

write a default text into the text field, the keyword *MULTILINE* would allow line breaks and the number 2 sets the border width to 2. It is, of course, the responsibility of the programmer of the configuration to implement a meaningful interpretation.

2.4 Layout

We tried to design a 'text-based WYSIWYG' language. And the wiring of components is solved with a self-explanatory natural-language approach. That makes it more natural and easy to read.

2.4.1 Text-based WYSIWYG

The programming concept follows the way the result looks like: there are sections in the program like the *SEE* section (what one can see), the *ACT* section (how it acts) and the *OFFER* section (what services it offers).

```
THE Texteditor
  model AS Text
SEE
  toolbar AS Panel AT TOP
  name AS Textfield AT LEFT IN toolbar WITH SINGLELINE, "untitled"
  open AS Button AT LEFT IN toolbar WITH "Load"
  save AS Button AT LEFT IN toolbar WITH "Store"
  text AS Textfield AT CENTER WITH MULTILINE
ACT
  text displays model
  open, name loads model
  save, name stores model
DATA
  caption = "Editor".
```

The arrangement of the components is specified with the keyword *AT* and the constraints *LEFT*, *TOP* and *CENTER*. There are several others, and one can even specify own positioning functions.

2.4.2 Positioning Functions

Components can use positioning functions to obtain more complex positions than the ones provided by default (*LEFT*, *TOP*, *RIGHT*, *BOTTOM*, *CEN-*

TER). For example, *Random* would place the components randomly. Positioning functions are specified in the configuration of the compiler.

2.4.3 Containers

Like one can see in the example code of *Texteditor* above, the visible instances are placed within other components. With the keyword *IN* one can specify the container of a visible instance. If it is omitted like for *toolbar* and *text*, intuitively, the root component is the container. Every Kulula component with at least one visible component is a visible component itself again. Non-visible components like *text* can contain only non-visible components.

3 Abstract Wiring

Since the components are combined to a new, more powerful unit, the relations between the components have to be specified in a way. Again, to reduce complexity, we tried to follow the event-handler pattern.

3.1 Composing and Supplement

Since the event-handler pattern is not powerful enough, and since we figured out that it is not possible to reduce overall complexity more than the complexity which is the result of the applications nature, we decided to emit a skeleton in another general purpose language to provide the full range of possibilities to implement the fine tuning of the wiring. With this approach, we split up the development process into two steps:

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

3.2 Event-Handler Pattern

The event-handler pattern separates cause and response. In Kulula, components raise events that are handled by other components's handlers. The Kulula compiler produces handler procedures and subscribes them to events. How it works in detail is part of the output language, and is explained in the appendix.

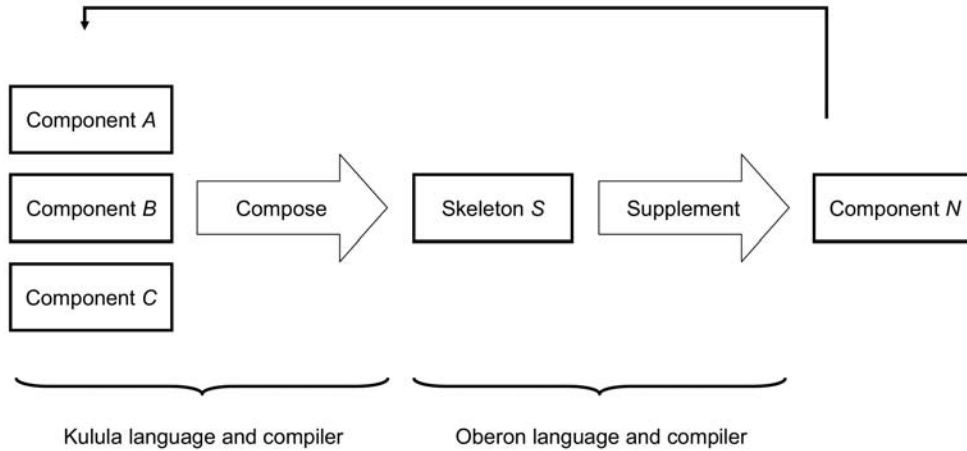


Figure 1: 2-phase design cycle.

3.3 Roles

Normally the user of a component does not understand the different events raised and the handlers offered completely, but he knows the abstract role between them. Each Kulula component offers (and defines respectively) such roles.

3.3.1 Connect With Wiring Contracts

For example in a model-view pattern[10], the text field component displays the text model component. In Kulula, this would look like the following:

```

THE Editor
  model AS Text
SEE
  textfield AS Textbox WITH MULTILINE

```

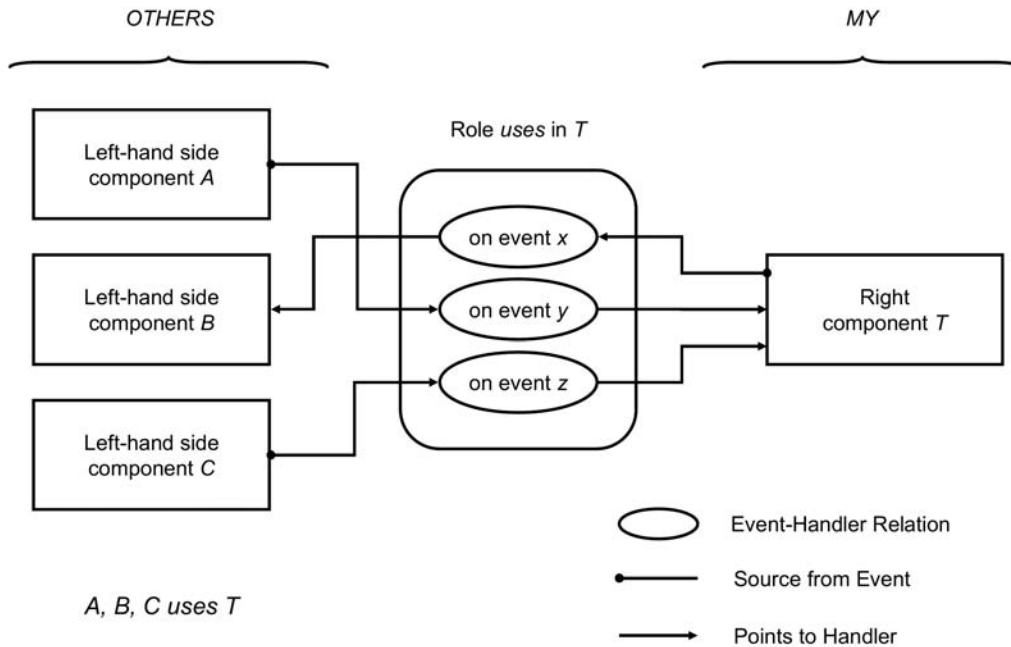


Figure 2: Schematic illustration of roles in Kulula

ACT

```
textfield displays model.
```

Such a wiring contract defined in the *ACT* section consists of a left-hand side component, a role name and a target. The BNF is the following:

```
WiringContract ::= IdentList Ident Ident.
```

```
IdentList ::= Ident { "," Ident }.
```

The left-hand side component might be a list of components, alternatively. This is useful, if we want to connect more than one component for a role. A good example is the model loading role:

```
THE Texteditor
  model AS Text
SEE
```

```

name AS Textfield WITH "Untitled"
open AS Button WITH "Load"
ACT
  open, name loads model.

```

The implementor of *Text* model already knows that for the *load* action a parameter with the filename is mandatory. But maybe he is not aware that there are components that do not have the ability to raise an event and to pass an argument at the same time. Fortunately, the two components together have all the events and arguments necessary.

3.3.2 Role Definition

The role *displays* is defined in the *OFFER* section of the target component (the component on the right-hand side), and specifies a set of event-handler relations:

```

THE Text
...
OFFER
  displays
    OTHERS SetText WITH MY GetContent AS String ON MY Change
    MY SetContent WITH OTHERS GetText AS String ON OTHERS Change.

```

The definition above states that each time the text model changes due to, e.g., a file system change (*ON MY Change*), the text field should be updated (*OTHERS SetText*). In the other direction, always when the user modifies the text via the text field (*ON OTHERS Change*), the model's content should be adapted (*MY SetContent*). The keyword *MY* denotes the own component in which the role is defined, or in other words, *MY* is the right side of the wiring contract. The keyword *OTHERS* symbolises the components on the left side of the wiring contract. Kulula figures out which handler or event of what component is used.

3.3.3 Naming Convention

Since the role identifier is used between two components and it describes a complex action, role names should be verbs in the singular form of the third person in lower case letters, e.g. *she stores*, *he loads*, *he baptises*.

Additionally, one can think about antonyms if the role has to be implemented in the other direction.

4 Dynamic Data

Inspired by the Bluebottle environment, we tried to handle data separately from the program logic. Therefore, data has its own section in each Kulula program.

4.1 Data Isolation

When we separate data and program logic we can simply replace and reassign values to properties of components. Components can get such data from the property list, from the given data in the *DATA* section, or with querying the indicated XML document. For example, the following is possible:

```
THE Example
SEE
  mybutton AS Button
DATA
  mybutton-caption="click me!"
  mybutton-height="150"
  caption="example application".
```

The *DATA* section contains short hand definitions of data keys. A short hand data definition consists of [*Ident* "-"] *Ident* "=" "" *Value* "" where the first optional *Ident* denotes the target component and the *Ident* after the dash specifies the property key. The *Value* may contain any valid string without the " sign.

4.2 Data Import

Another possibility is the indication of a XML document. Like that, we can assign language and region dependent data.

```
DATA
  FROM "myapp.xml"
```

The compiler tries to connect the available XML data with the properties of the components like it is done with the short hand definitions. Further, the necessary XML parsing procedures are emitted and the corresponding calls are written, as well. The example with the short hand data definitions from above would be translated to XML as following:

```
<?xml version='1.0' encoding='utf-8'?>
<example>
  <caption>example application</caption>
  <mybutton>
    <caption>click me!</caption>
    <height>150</height>
  </mybutton>
</example>
```

4.3 Internationalisation

A side effect of this function is, that one can specify

```
DATA
  FROM LOCAL "myapp.xml"
```

where *LOCAL* indicates that a locale appendix should be added at runtime. Such a locale appendix might be *DE* for German, for example (The runtime environment offers such a locale appendix on many platforms.) Kulula loads the corresponding data file ("*DEmyapp.xml* in the German version). (This functionality is not implemented in the current release)

5 Expandability

Thanks to the component technology, the constructs produced with Kulula can be composed to an other, better component. This process repeated recursively may offer some flexibility, but we have added the powerful concept of inheritance, too. Additionally, the language itself is extendable via the configuration of shortcuts and first class citizens.

5.1 Inheritance

One can extend existing components with the *OF* clause. All properties are inherited. And if the emitted and manually supplemented source is available, it is included:

```
THE ProfessionalEditor OF Editor IN MarvellousExamples
SEE
  assistant AS Paperclip WITH "Merlin".
```

The *IN* clause in the very first line indicates the group membership of the component. This group membership has no functional influence, but is necessary to map to name spaces like *MODULES* in Oberon, *packages* in Java or *namespaces* in *.Net*. (This functionality is not fully implemented in the current release. The scanner and parser accept the keyword, and the IR contains the modulename, but the code generator does not insert, nor replace the skeleton to the given module)

The following merge rules are applied:

Instance Overwriting An instance from the parent is taken, only if there is no instance with the same name.

```
THE Parent
  stays AS Something
SEE
  omitted AS Something.
```

```
THE Child OF Parent
SEE
  omitted AS ABetterThing.
```

results in

```
THE InheritanceResult
  stays AS Something
SEE
  omitted AS ABetterThing.
```

Unattached Contracts A parents contract is taken, only if all participants are inherited (not overwritten by the child).

Service Persistence All parents offerings are taken.

Data Overwriting A shorthand from the parent is taken, if the scope is not overwritten by an instance in the child.

5.2 Configuration

The configuration file is a valid XML document. It contains all output code snippets and definitions of first class citizens (shortcuts and components with property lists). One can modify and add new such definitions very easily. The compiler retrieves all its data from this file. After modifications, the compiler MODULEs need to be freed!

The XML has the following structure:

```
<?xml version="1.0"?>
<Kulula version="0"
    subversion="6"
    author="salam@student.ethz.ch">
<Types>
  <!-- The basic types - shorthands -->
  <Type wrap="Boolean" oberon="BOOLEAN"/>
  ...
  <!-- Kulula Foundation Classes -->
  <Type wrap="MediaPanel" oberon="MediaPanel.MediaPanel"/>
  <Type wrap="Text" oberon="KFCText.Text"/>
  ...
  <Type wrap="ToggleButton"
    oberon="KFCToggleButton.ToggleButton">
    <Property type="Autosize"
      place="Init"
      object="*"
      oberon="*.bounds.SetExtents( 80, 20 );"/>
    <Property type="Name"
      name="OFF"
      default="TRUE"
      place="Init"
```

```

        object="*"
        oberon="*.SetState(FALSE);"/>
    ...
    <Property type="Literal"
        place="Init"
        object="*"
        value="?"
        oberon="*.SetCaptionOn(&#x22;?&#x22;);"/>
    ...
    <Property type="Numeric"
        place="Init"
        object="*"
        value="?"
        oberon="*.bounds.SetWidth( ? );" />
    ...
</Type>
...
</Types>
<!-- Reserved words -->
<Reserved>
    <!-- Reserved since they are keywords of the language -->
    <Keyword name="THE"/>
    <Keyword name="OF"/>
    ...
    <!-- Reserved for future implementations, or
        Oberon expressions -->
    <Future name="ARTIFICIAL"/>
    <Future name="PROCEDURE"/>
</Reserved>
<!-- The trailer is used to specify standard commands
    that should be appended to the Oberon Module. -->
<Trailer object="xyz"
    value="S.Free xyz ~\nxyz.Start~\n
        xyz.Stop~\nPC0.Compile /s xyz.Mod~\n
        PET.Open xyz.Mod"/>
<!-- Positioning functions -->
<Positioning>
<Function name="Random" Type="Oberon">

```

```
(* Returns a randomly chosen area
   for a Kulula component *)
PROCEDURE KLLRandom( o: ANY; i : INTEGER )
                  : WMRectangles.Rectangle;
...
</Function>
</Positioning>
</Kulula>
```

5.3 Platform And Language Independence

To be able to use Kulula programs on several platforms with different output languages, the Kulula source does not contain any fragments of origin code. Even the event-handler calls are abstract. The language dependent code snippets are specified in the configuration file. The compiler loads the corresponding XML attribute (in our case the `oberon` attribute). Like that it is even possible to include snippets in other languages in the same file and one can implement a cross-platform compiler.

6 Conclusion

This is the first version of the language Kulula. And there are some possibilities to improve the language. Overall, we are no more that enthusiastic like before the implementation of the prototype. The language is useful for specific application like GUI production. For now, it is not possible to see the real impact of the language unless more foundation components are implemented.

Another aspect is that Kulula would be, maybe, more interesting in a multi-programming language environment like the *Microsoft .Net* architecture where one can implement a compiler that emits intermediate code, and the supplementation is done in other languages like *C#*, *Oberon.net* or *Eifel.net*.

The most interesting characters of Kulula are the role approach and the positioning functions for placement problems. A similar approach like the role relation is used in *TinyOS*[6]. In *C#*, one could translate these concepts with delegates for layout methods.

6.1 Problems

6.1.1 Data Redundancy And Conflicts

Because of the coexistence of properties and data definition with scopes, it is ambiguous what value should be assigned. Consider the following program:

```
THE AmbiguousProgram
SEE
  mybutton AS Button WITH "To be!"
DATA
  mybutton-caption="Or not to be!".
```

A possible workaround would be to define on definition stronger than the other. Normally the definition from the *DATA* section is more up-to-date. Therefore, it would make sense to overwrite the value from the *WITH* clause.

6.1.2 Wiring And Parameters

Another problem is the neat connection of parameters. There should be a possibility to rename and reconnect parameters within the rule definition. The current attempt with the *WITH* clause is insufficient.

6.2 Possible Improvements

6.2.1 Semantic Checks And *BUT NOT* Clause

The wiring contract offers a great possibility to check if all necessary events and handlers are available. Due to the fact that this events and handlers are specified in the supplemented output source, another parsing step would be necessary. Another interesting functionality would be, that one can loosen the contract with a *BUT NOT* clause after the wiring contract. It would not be a contract anymore, but an individual deal:

```
WiringDeal ::= IdentList Ident Ident [ "BUT NOT" IdentList ]
```

The list after *BUT NOT* includes the names of the events and handlers not provided by the left side.

6.2.2 Foundation Components

The current version of the compiler needs Foundation Components. Such Kulula components are not translated to Oberon code, but the information from the *OFFER* section is used to connect the other components with it, correctly.

Since at the beginning there are no Kulula components, one has to provide a wide range of foundation components (dummy Kulula programs) that have to necessary role and wiring information. We have written such dummy Kulula programs for the components necessary of the examples in this report, but with a larger collection of foundation components more complex problems can be solved.

6.2.3 More Language Independence

The compiler could be redesigned to support full language independence. With the current architecture, languages from different families are difficult or impossible to implement. Probably, one has to consider a redesign of the configuration file, as well.

6.2.4 Automatic Data Extraction

The current compiler tries to extract some information from the short hand data definitions and uses it in a copy-paste manner. Better would be an

attempt to create a new XML document with the data from the short hand definition.

6.2.5 Cross-Platform Compiler

A cross-platform compiler could be interesting to improve the impact of using Kulula in development. One has to compose the component once and one gets skeletons for different platforms and languages.

6.2.6 Generated Scanner And Parser

All parts of the prototype compiler, especially the scanner and parser, are implemented manually without the help of a generator tool. Maybe, one can make the compiler more correct using such tools.

6.2.7 Extended Property And Data Types

In this version, Kulula supports the three primitive data types *string*, *long* and *boolean*. It would be interesting to be able to assign whole record types and other data structures.

6.2.8 Integration Into A General Purpose Language

One should review the fact that compose and supplement take place in to different environments. Maybe, it would be interesting to integrate the language as compiler instructions, C# attributes or special Java comments.

A Compiler Design

The implemented prototype is a compiler that takes a Kulula source file, opens, scans and parses it, and builds up an internal data structure of the program. After that it checks some semantic properties and finally, it generates the Oberon source (skeleton) and writes it to a file. In the following subsections, the different phases are shortly described.

A.1 Scanning And Parsing

First, the scanner reads in all the tokens and then, it starts the parser. After that, the parser generates the internal data structure. To see the correct interpretation, the intermediate representation of the program is printed to the *log* window. The scanner is a derivate of the existing scanner, part from the original AOS compiler. The parser is implemented as a finite state machine. No look-ahead is necessary.

A.2 Semantic Checks

The following semantic checks are performed, so far:

- Type check: are types defined? In configuration file? Or is there a Kulula source for it?
- Data reference: exists the XML file indicated?
- Declaration check: are all idents declared as hidden or visible instances?
- Reserved keywords check: are ident names reserved keywords?
- Role check: are indicated roles defined by the right side of the contract?

In this phase the configuration gives the information for shortcut types and first class citizens. Some of the types will start a recursive compilation since their source is not compiled so far. Another recursive compilation is caused by inheritance to enhance the IR.

There are other checks that are not performed, but should be considered in another release:

- Property validation: is the indicated property applicable for this type?

- Contract satisfaction tests: are all procedures requested by one side, implemented by the other side?

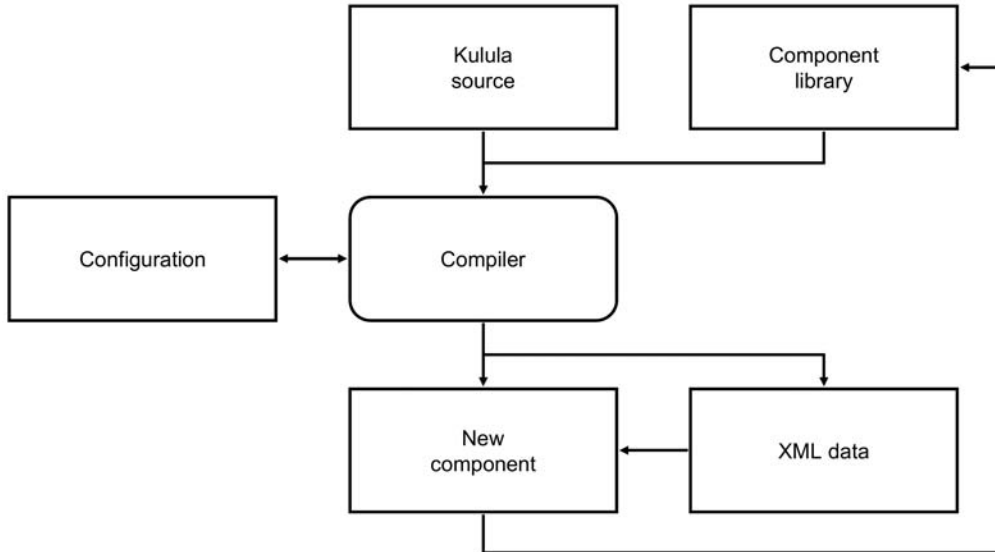


Figure 3: Compilation strategy

A.3 Code Generation

The code is generated by iterating the IR with the information from the configuration file. First, a given skeleton prolog is emitted. In a second step, the declarations of the containing components are generated. After that, the initialization phase sets up all the components:

- their position is calculated,
- property lists are translated to assignments and
- the short hand data definitions are implemented or

- the XML queries are connected to the XML sources.
- Finally, the event handlers are generated.

A.4 Runtime Services

Some functionalities of the components are outsourced to the Kulula module, since they are the same for all.

A.4.1 XML Queries

The XML queries in the components are implemented as procedure calls with the query as parameter and the return value as result. There are three types of return values: *strings*, *longs* and *booleans*. The given XML file is parsed and the key tag is read out. The result is returned in the specific form. Most of the properties can be handled like this.

A.4.2 Component Monitor And Control

Every component is subscribed to a component list and gets a unique *index* besides of its component *name*. A monitor procedure shows the current list of running components and one can stop a specific running component directly. Normally terminated components are unsubscribed by themselves. Kulula offers the three basic services: *subscribe*, *stop* and *unsubscribe*.

A.5 Modules Implemented

The implementation of the compiler prototype is described in this subsection, briefly. The following Oberon MODULES are implemented:

- *Kulula.Mod* contains the AOS commands, such as *Compile*, and the public API (runtime services).
- *KululaScanAndParse.Mod* contains the scanner and parser for the compiler.
- *KululaIR.Mod* defines the intermediate representation data structure and offers some tools for it.
- *KululaXML.Mod* offers the XML handling functionality.

- *KululaTools.Mod* offers several common tasks tools.
- *KululaConfig.Mod* contains the configuration reader, writer and the corresponding dictionary.
- *KululaCodeGenerator.Mod* contains the Active Oberon skeleton generator. It contains the procedure with the semantic checks as well.

B Component Skeleton

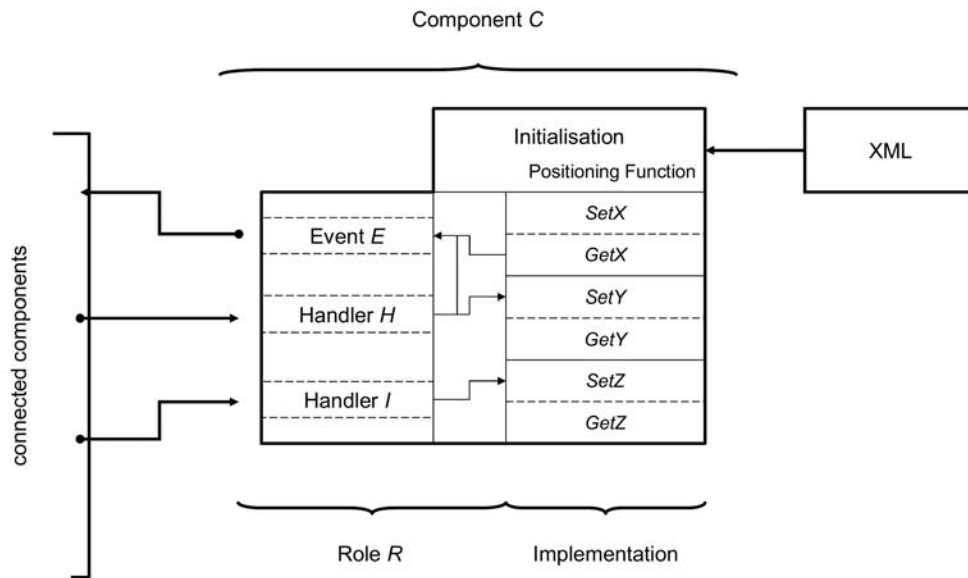


Figure 4: A schematic illustration of a component.

In this section, we will describe the implementation of a component in Bluebottle. The implementation depends on the language, the operating system architecture and the API environment, of course. Hence, the following is just a short description.

B.1 Hierarchy

Every component consists of an Active Oberon object[2][3]. Either it is from type *Kulula.VisibleComponent*, or *Kulula.HiddenComponent*. An inherited component (*OF* clause) will inherit from them as well.

B.2 Declarations

The hidden and visible components within the new Kulula component are declared first as attributes of the object. The real Oberon type is used, but the same name like in the Kulula program source.

B.3 Initialisation

The component is initialized with the *&Init* procedure, Here, all components contained are initialized recursively with their properties described in the *WITH* clause, in the *DATA* section or an extern XML source. Additionally, all the event-handlers are subscribed to the corresponding events.

B.4 Event-Handlers

For every relation in the wiring contract a separate event-handler procedure is generated. Thanks to the power of the general purpose programming language (in our case *Oberon*) it is possible to supplement them with individual operations.

B.5 Positioning Functions

The positioning functions are implemented as procedures with given interface:

```
PROCEDURE KLLZickZack( o: ANY; i: INTEGER ): WMRectangles.Rectangle;
```

The first argument (*o: ANY*) is the object that should be placed. The second argument (*i: INTEGER*) holds the index where the object is placed at. The return value (*WMRectangles.Rectangle*) contains the rectangle where the component should be placed at within the container.

SEE

```
first AS Label AT ZickZack
second AS Panel AT ZickZack
...
nth AS Text AT ZickZack
```

will result in the following procedure call ordering in the skeleton:

```
KLLZickZack( first, 0 );
KLLZickZack( second, 1 );
...
KLLZickZack( nth, n );
```

C Runtime Environment

Kulula components are compiled to *Active Oberon Objects*. After that an instance of such an object is instantiated each time another component needs a version of it.

C.1 Instantiation

During instantiation of a component all its subcomponents are instantiated in the order they are given in the source code (from the top to the bottom). Like that components are instantiate recursively if the subcomponents consist of subcomponents again:

C.1.1 Loading

In the meanwhile, the properties from the property list and the external XML source are applied to the component and the layout method is executed.

C.1.2 Wiring

At the end the local handlers are connected to the events of the subcomponents.

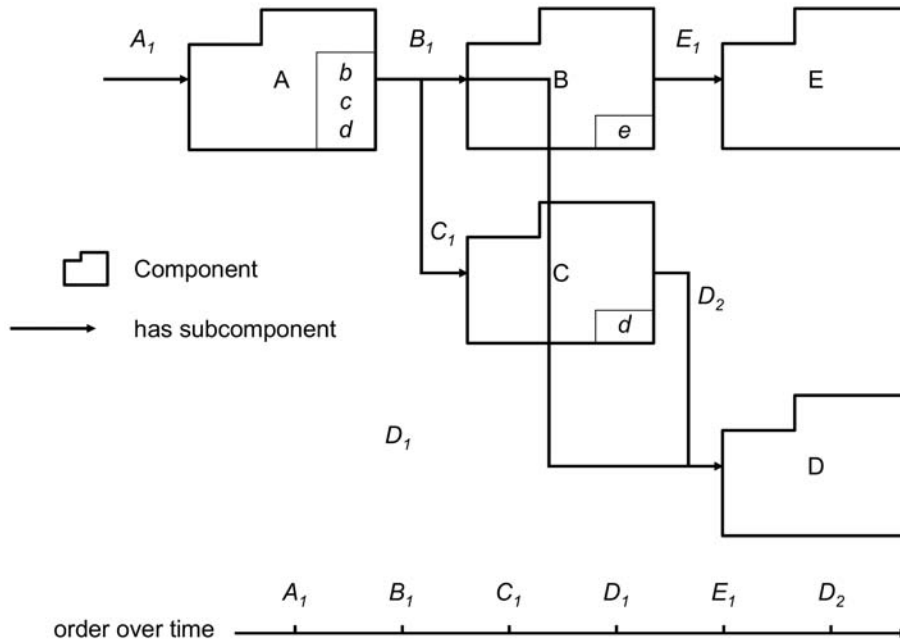


Figure 5: Instantiation over time.

C.2 Event-Handler Relation

All widgets in *Bluebottle* and all Kulula components offer a list of events and handlers available. The lists provide information for the semantic checker. If everything is correct, the handler is subscribed to the event.

C.2.1 Normal Scenario

In a normal case, a component's event raises the attached handlers. After that, the call stack shrinks and the event is handled.

C.2.2 Loop-Protection And Locking

Since Kulula roles support duplex wiring, a call loop is possible. To protect from an infinite loop, a semaphore bit for each handler is maintained.

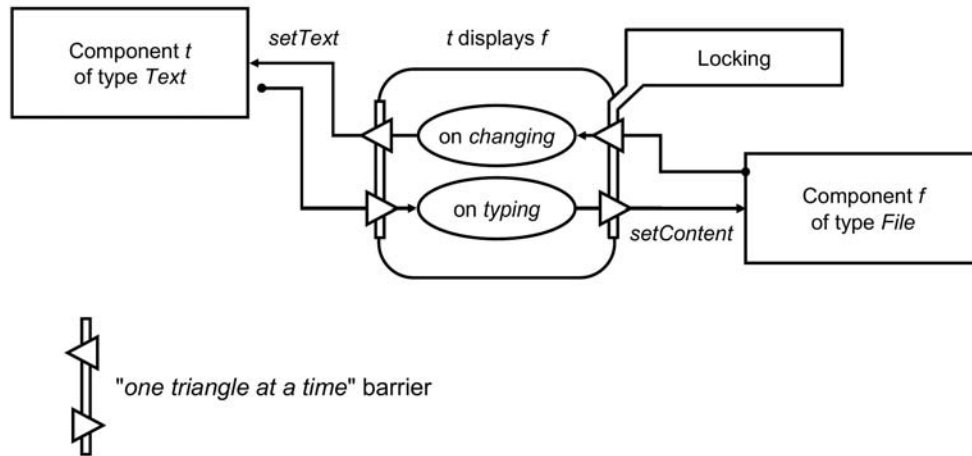


Figure 6: Loop-protection with locking.

C.3 Dynamic Data Retrieval

In an optional step, if the runtime system could reflect the component, the external XML source is applied to the rest of the data fields. This is a possible extension in a future version of Kulula.

C.3.1 Reflection

A possible strategy for obtaining applicable properties is to reflect procedures with the *Get* or *Set* prefix. The second part of the procedure name is interpreted as property name. The *dynamic data loader* applies the value from the XML file with the very same name like the component. Anyway, this is not implemented, yet.

C.4 Termination

Kulula components provide a finalize procedure to release, or finalize, all its subcomponents, as well. The finalize method is called by the Bluebottle system, if the system is shut down or a visual component is closed with a mouse click.

C.4.1 Normal Termination

Normal termination is raised by the runtime system provided by the Bluebottle system. Possible reasons are system shut downs and manual closing of windows.

C.4.2 Forced Termination

With *Kulula.Stop Componentname/instance-index*, the component is forced to stop. After that it may be impossible to run any components that have the component stopped as a subcomponent.

D Step-by-step Example: A Simple Text Editor

D.1 Compose

In this section we want to program a tiny text editor with some extended functionality: the save button (label is *Store*) should be relabeled with *Store !*, if the model isn't propagated to persistent memory like to a local file, or to a remote file on a FTP server. Therefore, we write the following text to a file named *Texteditor.Kulula*, first:

```
THE Texteditor
  model AS Text
SEE
  toolbar AS Panel AT TOP
  name AS Textfield AT LEFT IN toolbar WITH SINGLELINE, "untitled"
  open AS Button AT LEFT IN toolbar WITH "Load"
  save AS Button AT LEFT IN toolbar WITH "Store"
  text AS Textfield AT CENTER WITH MULTILINE
```



Figure 7: Screenshot of sample program *TextEditor*.

```
ACT
  text displays model
  open, name loads model
  save, name stores model
DATA
  width = "300"
  height = "180"
  caption = "Editor".
```

D.2 Generate skeleton

After that, we have to generate the skeleton in Oberon with the command *Kulula.Compile Texteditor* . The binary is compiled automatically, and on the console the the following output (or something similar) should be written:

```
Kulula, Version 0.2
Configuration loaded.
```

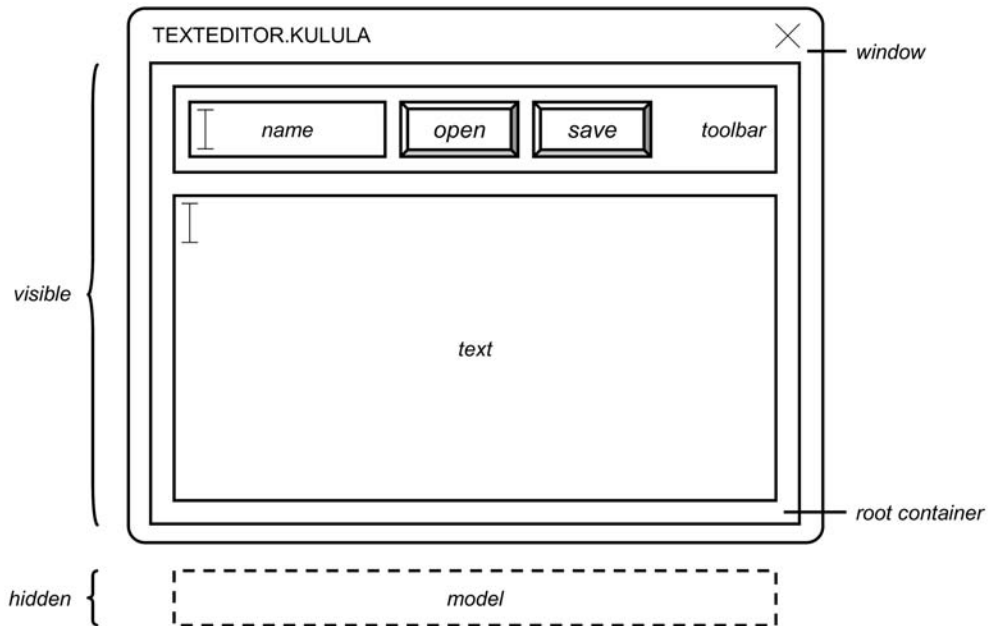


Figure 8: Schematic layout of sample program *TextEditor*.

```

Compile Texteditor.Kulula...
Skeleton generation was successful!
Texteditor.Mod compiling Texteditor 2584 done
  PET.Open Texteditor.Mod ~
  Texteditor.Start ~
  Texteditor.Stop ~

```

D.3 Run

Now, we can start the component with *Kulula.Start Texteditor*. A window with the visual component in it will be shown.

D.4 Supplement

But since we want to improve the program with the exclamation mark re-labeling functionality, we have to supplement the *Texteditor* component in the *Texteditor.Mod* file.

Well, there are two states: a model is propagated, or not. So, we will add a relabel statement in the handler procedure named *OnChangeText*:

```
save.Caption.SetAoC("Store !");
```

And vice-versa, we extend the handlers *OnChangeModel* and *OnClickSave* with

```
save.Caption.SetAoC("Store");
```

D.5 Compile

Since we have changed the components source, we have to compile the *MODULE Samples*, first:

```
PC.Compile Texteditor.Mod~
```

D.6 Run

Now, we can start the improved component with *Kulula.Start Texteditor* and check out if it works.

E More Examples

In the following section, some examples are provided to show specific aspects of Kulula.

E.1 Control Nightmare introducing Placement Functions

```
THE ControlNightmare  
SEE  
one AS Panel AT TOP
```

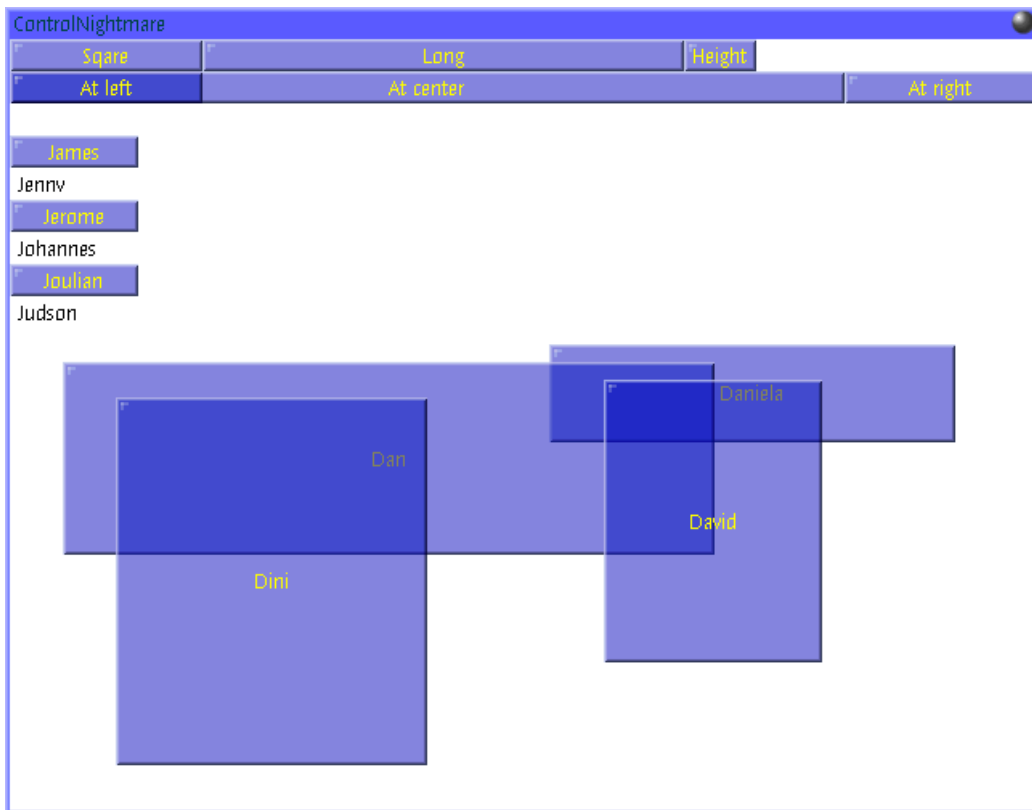


Figure 9: Many widgets with the *Random* placement function - a *Control-Nightmare*.

```
two AS Panel AT TOP
three AS Panel AT TOP
four AS Panel AT CENTER
five AS Panel AT BOTTOM WITH 300, 300
```

```
adriana AS Panel IN four AT LEFT
aline AS Panel IN four AT LEFT
andrea AS Panel IN four AT LEFT
andresa AS Panel IN four AT LEFT
anna AS Panel IN four AT LEFT
```

```

chantal AS Button IN one AT LEFT WITH "Sqaare", 120, 120
chauhop AS Button IN one AT LEFT WITH "Long", 300, 45
chiara AS Button IN one AT LEFT WITH "Height", 45, 450

benno AS Button IN two AT RIGHT WITH "At right", 120, 40
boris AS Button IN two AT CENTER WITH "At center", 120, 40
bruno AS Button IN two AT LEFT WITH "At left", 120, 40

daniela AS Button IN five AT Random WITH "Daniela"
dan AS Button IN five AT Random WITH "Dan"
david AS Button IN five AT Random WITH "David"
dini AS Button IN five AT Random WITH "Dini"

james AS Button IN adriana AT TOP WITH "James"
jenny AS Textfield IN adriana AT TOP WITH "Jenny", SINGLELINE
jerome AS Button IN adriana AT TOP WITH "Jerome"
johannes AS Textfield IN adriana AT TOP WITH "Johannes", SINGLELINE
joulian AS Button IN adriana AT TOP WITH "Joulian"
judson AS Textfield IN adriana AT TOP WITH "Judson", SINGLELINE.

```

E.2 FilenamePicker and BetterFilenamePicker introducing Inheritance

THE FilenamePicker

SEE

```

filename AS Textfield AT LEFT WITH SINGLELINE, "untitled"
choose AS Button AT LEFT WITH "Choose..."

```

DATA

```

height = "100"
width = "250".

```

THE BetterFilenamePicker OF FilenamePicker

SEE

```

delete AS Button AT LEFT WITH "Delete!"
info AS Button AT LEFT WITH "Get Fileattributes"

```

DATA

```

width = "400".

```

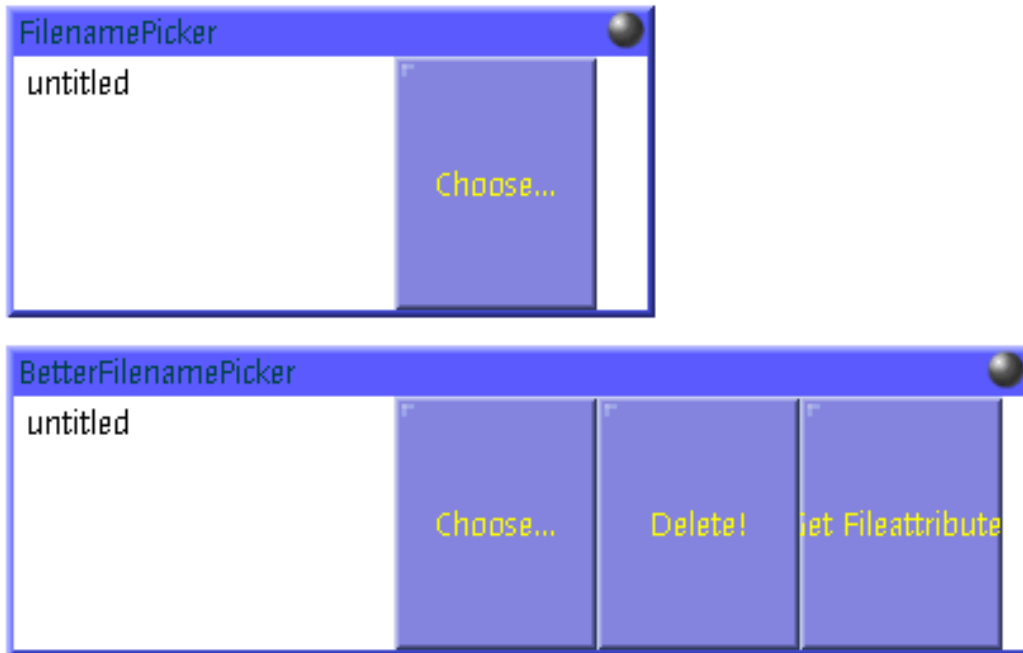


Figure 10: The *BetterFilenamePicker* extends the *FilenamePicker* with some widgets.

E.3 Mediaplayer in 40 Words Only

THE Mediaplayer

SEE

movie AS MediaPanel AT TOP

toolbar AS Panel AT BOTTOM

name AS Text IN toolbar AT LEFT WITH SINGLELINE, BORDER

play AS ToggleButton IN toolbar AT LEFT WITH "Play", "Stop"

ACT

name denotes movie

play controls movie.

THE MediaPanel OF KululaFoundationClass

OFFER

feeds WHERE

MY SetFilename WITH OTHERS GetFilename AS String ON OTHERS onChange



Figure 11: A simple *Mediaplayer* described in just 40 words.

```
controls WHERE
```

```
MY Play WITH OTHERS GetState AS Boolean ON OTHERS onClick
```

```
denotes WHERE
```

```
MY Open WITH OTHERS GetAsString AS String ON OTHERS text.onTextChanged.
```

E.4 SubContainer, et al. introducing Reuse

```
THE SubContainer
```

```
SEE
```

```
left AS MyLeftThing AT LEFT
```

```
middle AS MyMiddleThing AT LEFT
```

```
right AS MyRightThing AT LEFT.
```

```
THE MyLeftThing
```

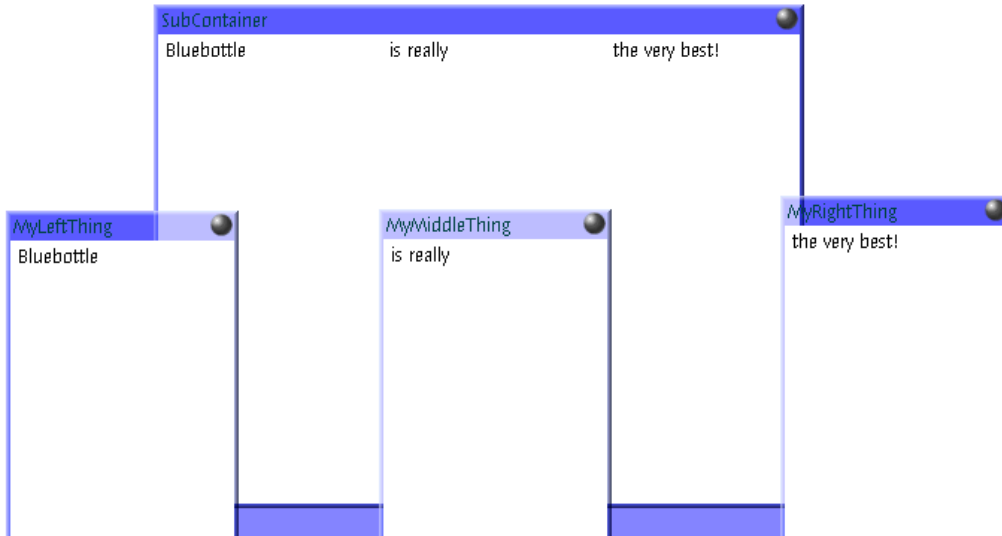


Figure 12: The windows with the *SubContainer* (consisting of three other embedded Kulula components) and the separately started components *MyLeftThing*, *MyMiddleThing* and *MyRightThing* in the front.

SEE

```
myleftthing AS Textfield AT ALLOVER WITH "Bluebottle", SINGLELINE, 150
DATA
width = "150".
```

THE MyMiddleThing

SEE

```
mymiddlething AS Textfield AT ALLOVER WITH "is really", SINGLELINE, 150
DATA
width = "150".
```

THE MyRightThing

SEE

```
myrightthing AS Textfield AT ALLOVER WITH "the very best!", SINGLELINE, 150
DATA
width = "150".
```

F Language Specification In BNF

```
Kulula ::= "THE" Ident
        [ "OF" Ident ]
        [ "IN" Ident ]
        Declarations
        ".".

Ident ::= Letter { Letter | Digit }.
Letter ::= "A" | "B" | "C" | ... | "x" | "y" | "z".
Digit ::= "0" | "1" | "2" | ... | "9".
Declarations ::= { HiddenInstance }
                [ VisibleInstances ]
                [ Behaviour ]
                [ Offerings ]
                [ Data ].

HiddenInstance ::= Declaration [ "WITH" Properties ].
Declaration ::= IdentList "AS" Ident.
IdentList ::= Ident { "," Ident }.
VisibleInstances ::= "SEE" VisibleInstance
                   { VisibleInstance }.

VisibleInstance ::= SingleDeclaration
                 [ "IN" Ident ]
                 [ "AT" Position ]
                 [ "WITH" Properties ].

SingleDeclaration ::= Ident "AS" Ident.
Position ::= "LEFT"
            | "TOP"
            | "RIGHT"
            | "BOTTOM"
            | "CENTER"
            | "AUTO"
            | Coordinates
            | Ident.

Coordinates ::= "(" Numeric "," Numeric ")"
              [ "," Numeric "," Numeric ] ".

Numeric ::= Digit { Digit }.
Properties ::= Ident
            | Numeric
```

```

    | String.
String ::= "" { Ident | " " | Special } "".
Behaviour ::= "ACT" { WiringContract }.
WiringContract ::= IdentList Ident Ident.
Offerings ::= "OFFER" Role { Role }.
Role ::= Ident "WHERE" Pair { Pair }.
Pair ::= Possessor String [ Parameters ] "ON"
        Possessor String [ Parameters ].
Possessor ::= "MY"
            | "OTHERS".
Parameters ::= "WITH" Possessor Declaration
              { ", " Possessor Declaration }.
Data ::= "DATA" ( Shorthands | XMLReference ).
Shorthands ::= Shorthand { Shorthand }.
Shorthand ::= [ Ident "-" ] Ident "=" String.
XMLReference ::= "FROM" [ "LOCAL" ] String.

```

Acknowledgment

I would like to thank Thomas Frey for his extraordinary first level Bluebottle support. I also thank Prof. Gutknecht for the generous freedom he gave me to design this language and Prof. Pieter de Villiers from the Stellenbosch University for the inspiring discussions.

References

- [1] T. Frey. Bluebottle: A Thread-safe Multimedia and GUI Framework for Active Objects. PhD thesis at *Swiss Federal Institute of Technology*. 2005.
- [2] P. Reali. Active Oberon Language Report. 2004.
- [3] P. Muller. The Active Object System Design and Multiprocessor Implementation. PhD thesis at *Swiss Federal Institute of Technology*. 2002.
- [4] N. Wirth, J. Gutknecht. Project Oberon. Addison-Wesley, 1992.
- [5] T. Reenskaug, et al. Working with objects: The OOram Software Engineering Method, 1995.
- [6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems, In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003.
- [7] Open Services Gateway Initiative. About the OSGi Service Platform, Technical Whitepaper. 2004.
- [8] H. Cervantes, R. S. Hall. Beanome: A Component Model for the OSGi Framework. In *Proceedings of the Software Infrastructures for Component-Based Applications on Consumer Devices Workshop*, 2002.
- [9] Macromedia Flash. ActionScript. www.macromedia.com
- [10] Gamma, E., et al., Design Patterns - Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.