

## Appendix A The Koala Language

This appendix defines the Koala language as of August 2003. Features that have been removed are still defined in this appendix but are clearly marked obsolete. This is not an introduction into Koala: familiarity with the language is assumed. Note that the Koala compiler may impose additional limitations on the use of the language.

Section A.1 explains the basic concepts of Koala. Section A.2 defines the lexical syntax. Section A.3 defines the syntax and well formedness of interface definitions, section A.4 of component definitions, and section A.5 of data type definitions. Section A.6 provides naming conventions, while section A.7 lists a non-optimizing code generation scheme. Section A.8 ends the appendix with concluding remarks.

For a proper interpretation of the syntax diagrams, the following rules hold:

- Every line segment is reachable.
- A line segment with an arrow can only be traversed in that direction.
- A vertical line segment can be traversed in either upward or downward direction.
- If a horizontal line segment can be traversed in rightward direction, it is forbidden to traverse it leftward direction.

### A.1 Concepts

The Koala language is based upon the following concepts:

- An *interface* is a small set of semantically related elements, which can be functions, parameters or constants. An *interface definition* describes the syntax and semantics of these elements, and is a unit of specification. An *interface instance* provides access to the functionality of a component, or it specifies the use of (external) functionality by a component.
- A *component* is a set of files in a single directory. A *component definition* describes the provided and required interfaces and the internal structure of the component, and is a unit of reuse. A *component instance* is a component as compiled and linked into a running system. A *basic component* is a component without subcomponents. A *configuration* is a component without interfaces on the border.
- A *module* is a unit of code; put differently, there is no code outside of modules. A module is contained in a component. Note that the module is *not* a unit of reuse, but the containing component *is*. The module implements and uses interface instances connected to the module. The implementation can be chosen per interface element: in C or in Koala expressions.

- A *cable* – or *interface binding* – connects an interface to another interface, an interface to a module, or a module to an interface.
- A *switch* connects a list of interfaces to one of a specified number of lists of other interfaces, depending on the evaluation of an expression. Switches are used to implement compile-time and run-time binding.
- A *fiber* – or *function binding* – implements a single function in a module as a Koala expression that can be partially evaluated by the Koala compiler, thus providing room for compile-time optimizations.
- The *repository* is the place where all component, interface and data type definitions are stored.

Koala targets C as implementation language.

## A.2 Lexical Syntax

The lexical syntax of the Koala component and interface definition language determines how to split a stream of characters into a sequence of tokens. Unless specified otherwise, the longest match is made.

- A non-empty sequence of space (ASCII code 32), tab (9), carriage-return (13) and line feed (10) characters is treated as *white space*. White space separates tokens but is further ignored during parsing.
- There are two forms of *comment*: between */\** and *\*/* brackets, and following *//* up to the first new line (line feed on most platforms) character. Comment is treated as white space and is further ignored during parsing. Bracketed comments may not be nested.
- An *identifier* is a sequence of letters ('a'..'z', 'A'..'Z'), digits ('0'..'9') and the underscore character ('\_'), starting with a letter. Keywords as defined below are not valid identifiers. Note that Koala is case sensitive.
- The following words are Koala *keywords*: **addressable**, *apply*, **component**, **connects**, **const**, **contains**, **except**, **false**, **file**, **group**, *handle*, **in**, **inline**, *instance*, **interface**, **koala**, **legacy**, *library*, **module**, *multi*, *no\_lib*, **no\_prefix**, **null**, **on**, **optional**, **otherwise**, **out**, **prefix**, **present**, **provides**, **requires**, *single*, **specials**, **switch**, **true**, **type**, **uses**, **using**, **void**, **within**. The keywords that have been made *italic* are obsolete.
- The following single character symbols are valid Koala *operators*: **;**, **?**, **|**, **^**, **&**, **<**, **>**, **+**, **-**, **\***, **/**, **%**, **!**, **~**. The following double character symbols are valid Koala *operators*: **||**, **&&**, **==**, **!=**, **<=**, **>=**, **<<**, **>>**. In any sequence of these characters, the longest match is preferred.

- Additional *symbols* recognized by Koala are: ‘(’, ‘)’’, ‘{’, ‘}’, ‘;’, ‘=’, ‘.’ and ‘:’.
- A *number* is a non-empty sequence of digits (‘0’.. ‘9’, see below how to represent hexadecimal numbers), optionally prefixed with ‘0x’ or ‘0X’. Numbers may be post fixed with ‘u’ or ‘U’ (for *unsigned*) or with ‘l’ or ‘L’ (for *long*) or both<sup>13</sup>. The notation is assumed to be:
  - *Decimal* if the number starts with ‘1’..‘9’.
  - *Octal* if the number starts with ‘0’. Only digits ‘0’..‘7’ are allowed<sup>14</sup>.
  - *Hexadecimal* if the number starts with ‘0x’ or ‘0X’. For digits > 9, the letters ‘a’..‘f’ or ‘A’..‘F’ can be used.
- A (normal) *string* is a sequence of characters enclosed in double quotes (“”). The double quote character itself cannot be included in a string. The back slash character (‘\’) is taken literally. An embedded new line is not allowed.
- A *code block string* is a sequence of characters enclosed in ‘@’ signs. Embedded new lines are allowed, nor are ‘@’ sign characters. Leading and trailing blank lines in the code block string are ignored, as is any leading and trailing white space of each line. Code block strings can be used wherever normal strings are used.

Data type definitions are C header files enriched with Koala annotations. There are three types of Koala annotations in these files:

```
/** koala type <identifier> */
/** koala group <identifier> */
/** koala using <identifier> */
```

Remarks:

- A sequence of space and/or tab characters can be used to separate the tokens in these annotations.
- Any other comment that starts with `/** koala` will result in an error.

Data type definitions are discussed in more detail in section A.5.

### A.3 The Interface Definition Language

Below, we provide syntax diagrams and well-formedness rules for the interface definition language (IDL). In the diagrams, rounded rectangles denote keywords, symbols or operators; plain rectangles denote non-terminals, identifiers or strings.

---

<sup>13</sup> The Koala compiler performs all integer calculations with 32 bit signed integers. Unsigned and long integer constants are therefore cast to a normal C ‘*int*’.

<sup>14</sup> Koala currently itemizes ‘09’ as octal ‘0’ followed by decimal ‘9’.

### A.3.1 Interface Definition

An *Interface Definition* (see Figure 79) defines a single interface type.

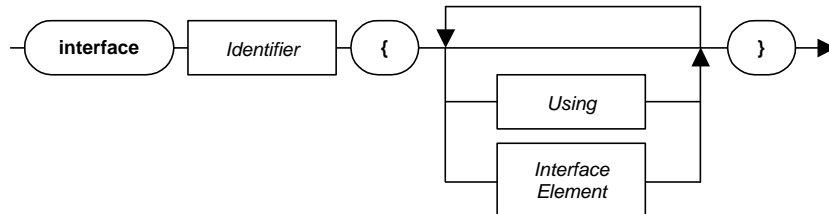


Figure 79. Interface Definition

The following rules apply:

- The identifier that follows **interface** defines the name of the interface type.
- This name must be globally unique: it may not be equal to the name of any other interface type, component type or data type (group) in the repository.

### A.3.2 Using

The *Using* clause (see Figure 80) makes additional data types accessible.

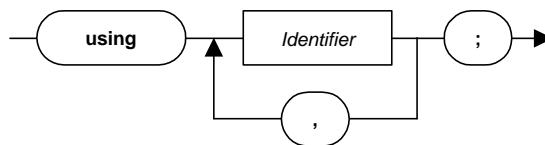


Figure 80. Using

The following rules apply:

- Each identifier listed in the using clause must represent a data type (see section A.5.1) or data type group (see section A.5.2) that is defined in the repository.
- The using clause is obsolete – there is no immediate use for it. All data types occurring in function prototypes are automatically made accessible, and components can specify using clauses themselves (see section A.4.4).

### A.3.3 Interface Element

An *Interface Element* (see Figure 81) specifies a function, an interface parameter or an interface constant.

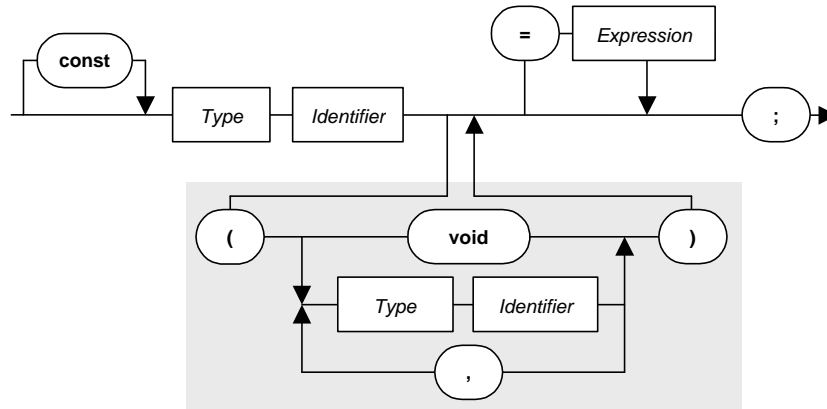


Figure 81. Interface Element

The following rules apply:

- If **const** is specified, then Koala will generate an error at an instance of this interface type if it cannot guarantee at compile-time that the element has a constant value.
- The *type* of the interface element should be valid (see section A.3.4).
- The identifier specifies the *name* of the interface element.
- This name should be unique for the interface definition.
- This name should also not be equal to the name of any data type or data type group in the repository.
- **Definition:** a *function* is an interface element that has a parameter list (shaded in Figure 81).
- The parameter list may be empty, which must be specified with **(void)**.
- If the parameter list is non-empty, then each parameter must have a valid type (see section A.3.4).
- Each parameter must have a name (the identifier) that is unique for the list of parameters, and that is not the name of a data type or data type group defined in the repository.
- **Definition:** an *interface parameter* is an interface element without a parameter list (shaded in Figure 81), and without an expression assigned to it.
- **Definition:** an *interface constant* is an interface element without a parameter list, but with an expression assigned to it.

- For an interface constant, it must be possible to calculate the assigned value by considering the interface definition alone.
- The expression may refer to other interface elements; in such cases the name of the element must be prefixed with the name of the interface type.
- It is not allowed to use an in-line expression (section A.4.21) in an interface definition.

### A.3.4 Type

A *Type* (see Figure 82) specifies the data type of an interface element or of one of its parameters.

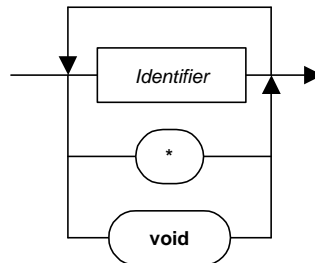


Figure 82. Type

The following rules and remarks apply:

- The sequence of identifiers, **void** and '\*' should be a valid C type (such as 'long int' and 'void \*')<sup>15</sup>.
- Remark: type constructors such as *struct*, *union* and *array* cannot be used.
- Koala treats each identifier (and also **void**) as a separate data type that should be defined in the repository (see section A.5.1), with the exception of 'char' and 'int', which are predefined in Koala.

### A.3.5 Interface Sub-typing

An interface type IA is said to be a *subset* of interface type IB if the following conditions are met:

- Every element of IA also occurs in IB, with the same name, the same type and – if present – the same parameter names<sup>16</sup> and types. C sub-typing is not permitted here, so 'char' and 'int' are different types.

<sup>15</sup> The current Koala compiler allows at most 2 identifiers. Also, it allows at most one '\*', which may follow either (but not both) of the identifiers.

- If an element of IB is assigned an expression in the interface definition, then either the corresponding element of IA must have an expression that results in the same value<sup>17</sup>, or it must not have an expression assigned to it at all.
- If an element in IA is declared **const**, then the corresponding element in IB must be constant, i.e. it must either be declared const as well, or it must have been assigned an expression that Koala can evaluate to a constant.

In formal type theory, IB is said to be a *sub-type* of IA.

## A.4 The Component Definition Language

Below, we provide syntax diagrams and well-formedness rules for the component definition language (CDL). In the diagrams, rounded rectangles denote keywords, symbols or operators; plain rectangles denote non-terminals, identifiers or strings.

### A.4.1 Component Definition

A *Component Definition* (see Figure 83) defines a single component type.

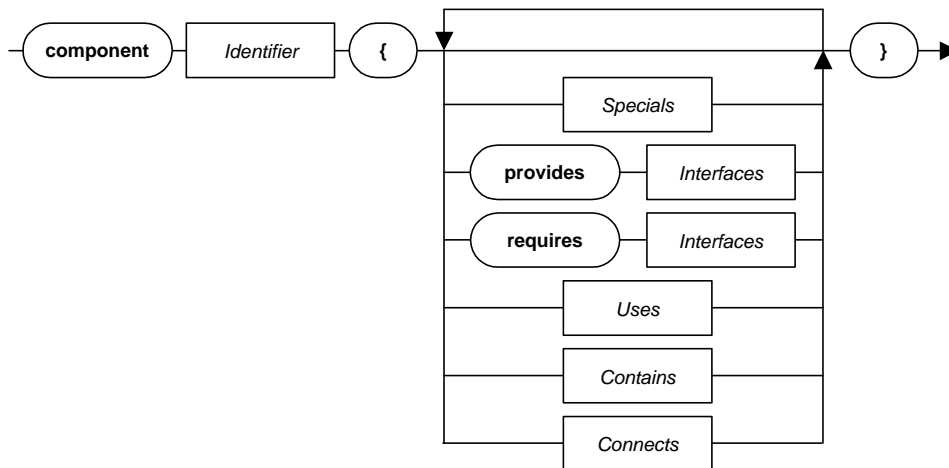


Figure 83. Component Definition

The following remarks apply:

- The identifier that follows the keyword **component** specifies the name of the component type.

<sup>16</sup> Note that the subtype relation is *implicit* in Koala. Parameter names are included in the subtype relation as an additional measure to prevent accidental subtyping.

<sup>17</sup> The Koala compiler requires that the expressions in the previous case be structurally equal.

- This name must be globally unique: it may not be equal to the name of any other component type, interface type or data type (group) in the repository.
- The **specials** clause declares some properties of the component (see section A.4.2).
- Interfaces declared after the keyword **provides** are called *provides interfaces*. They provide functionality to the environment of the component.
- Interfaces declared after the keyword **requires** are called *requires interfaces*. Through these the component requires functionality of its environment.
- The **uses** clause makes additional data types (and groups) accessible to all modules of the component (see section A.4.4).
- The **contains** clause declares the modules, subcomponents and internal interfaces of the component (see section A.4.5).
- The **connects** clause specifies the connections between interfaces and modules (see section A.4.8).
- Although the language allows the clauses mentioned above to be specified in any order, the order as specified in Figure 83 is recommended.

### A.4.2 Specials

The *Specials* clause (see Figure 84) defines properties of the component.

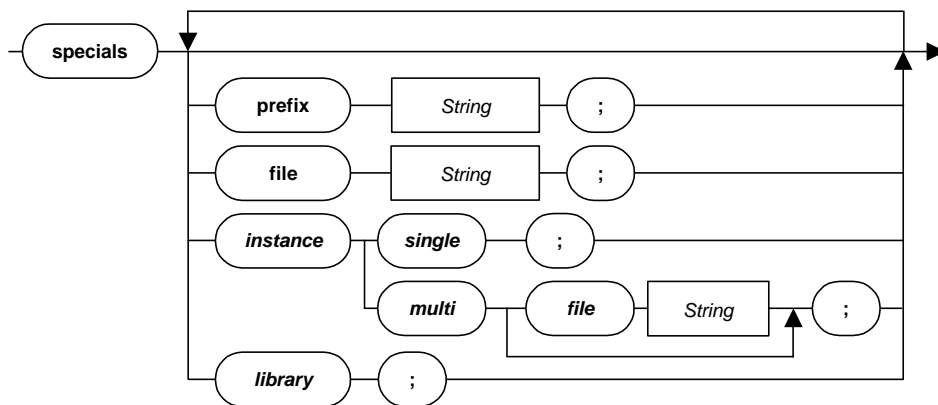


Figure 84. Specials

The following rules apply:

- The string that follows **prefix** specifies the *component prefix* (also known as the component *short name*) as used in the code generation.
- The component prefix must start with a letter and consist of a sequence of letters and/or digits (i.e., it may not be empty and should not contain ‘\_’).

- The component prefix must be globally unique: it may not clash with the prefix of any other component in the repository.
- If not specified explicitly, the prefix is assumed to be equal to the component type name (also known as the component *long name*).
- The string following **file** names the C file generated for the component if applicable.
- This string may not be empty, and it must include the extension (‘.c’).
- If not specified explicitly, the component prefix preceded by ‘\_’ and followed by ‘.c’ is used as name for the output file.
- The **instance** clause specifies whether the implementation of the component is prepared for multiple instantiation (MI). The default is single instantiation.
- For an MI component, the string following **multi file** specifies the name of the file where Koala generates the instance data.
- This file name may not be empty, and must include the ‘.c’ file type.
- The Koala compiler no longer supports multiple instantiation. The instance clause is documented here for historic reasons only.
- The **library** clause specifies that a library is to be generated for this component.
- The Koala compiler no longer supports the library clause. The library clause is documented here for historic reasons only.
- The **prefix**, **file**, **instance** and **library** clauses may appear in any order, but each clause may appear only once.

### A.4.3 Interfaces

A list of *Interfaces* (see Figure 85) declares provided, required or internal interface instances.

The following rules apply:

- The first identifier specifies the *type* of the interface.
- This interface type must be defined in the repository.
- The second identifier names the *instance* of the interface.
- The instance name must be unique within the component definition: it may not clash with the name of any other interface instance, subcomponent or module.

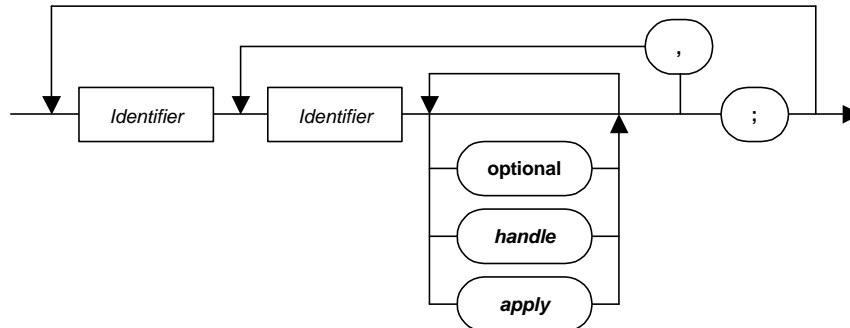


Figure 85. Interfaces

- An interface instance can be declared to be **optional**. If an interface is not optional, it is said to be mandatory (which is the default).
- An interface instance can be declared to be an **apply** interface, and/or it can be declared to have a **handle**. Both features are currently not supported by the Koala compiler.
- **optional**, **handle** and **apply** only relate to the interface of which the instance name directly precedes the specifiers.
- **optional**, **handle** and **apply** may occur only once, but can occur in any order.
- **handle** and **apply** are obsolete.

#### A.4.4 Uses

The *Uses* clause (see Figure 86) provides all modules in the component definition with access to the data types and data type groups specified.

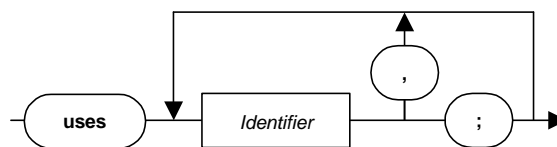


Figure 86. Uses

The following rules apply:

- Each identifier must refer to a data type or data type group that is defined in the repository.
- Note that modules have automatic access to all data types occurring in interfaces that they implement or use.

### A.4.5 Contains

The *Contains* section (see Figure 87) specifies the ‘parts-list’ of the component, i.e. it lists the modules, the subcomponents and internal interfaces of the component.

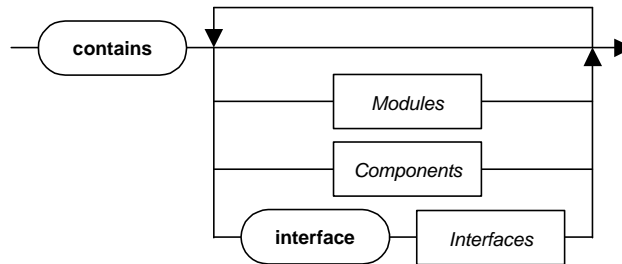


Figure 87. Contains

The following remarks apply:

- Modules contained in a component definition are the sole property of the component; reuse of the module outside of the component is not permitted.
- Components contained in a component definition are encapsulated instances of reusable component types. The instance can only be accessed through the component, but the reusable component type can have other instances in other components. However, a component can only be instantiated once in a single product, unless the component is capable of multi instancing.
- Interfaces declared after the keyword **interface** are called *private* or *internal interfaces*.

### A.4.6 Modules

A *Modules* list (see Figure 88) declares ‘glue’ (also known as ‘code’) modules.

The following rules apply:

- The identifier that follows the keyword **module** defines the name of the module.
- This name must be unique within the component definition: it may not clash with the name of any other module, subcomponent or interface instance.
- The string following **file** specifies the header file to be generated for this module.
- This string may not be empty, and should include the file extension (‘.h’).
- The default file name is an underscore, followed by the prefix of the component, an underscore, the name of the module and ‘.h’ (e.g. ‘\_c\_m.h’).

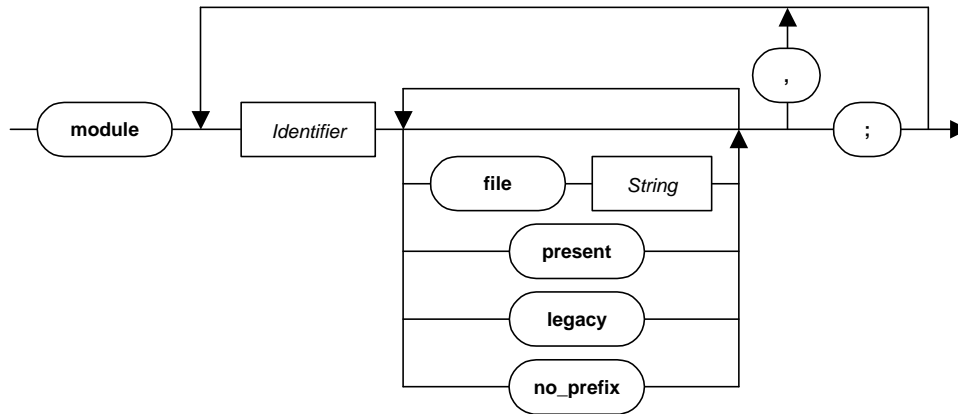


Figure 88. Modules

- The keyword **present** specifies that this module *must* be included when compiling a configuration that contains this component. By default, modules are only included when they are *reachable*.<sup>18</sup>
- The keyword **legacy** overrides the naming conventions in the generated header file, as explained in section A.7.
- The keyword **no\_prefix** has the same effect as **legacy**.
- The keywords **file**, **present** and **legacy/no\_prefix** only relate to the module of which the instance name immediately precedes the keywords.
- The keywords **file**, **present** and **legacy/no\_prefix** can be specified in any order, but each may occur at most once.

#### A.4.7 Components

A *Components* list (see Figure 89) declares subcomponents of a component.

The following rules apply:

- The first identifier specifies the *type* of the subcomponent.
- A component type with this name must be defined in the repository.
- The second identifier names the subcomponent *instance*.
- The instance name must be unique within the component definition: it may not clash with the name of any other subcomponent, interface, or module.

<sup>18</sup> Reachability is not discussed in this document.

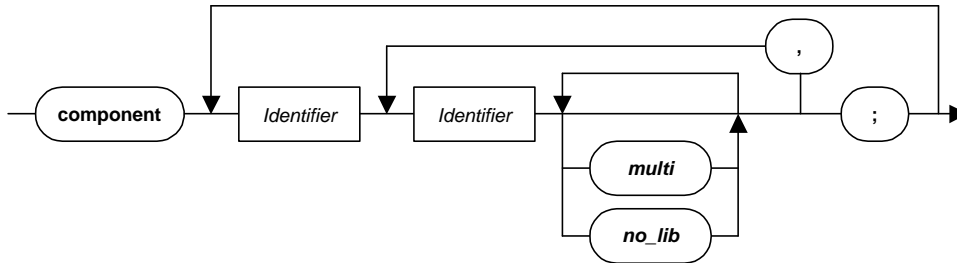


Figure 89. Components

- A preferred choice for the component instance name is the short name of the corresponding component definition.
- The keyword **multi** specifies that this subcomponent can be instantiated dynamically. Dynamic instantiation has never been implemented in Koala.
- If any of the subcomponents supports only single instantiation, then the component itself may support only single instantiation.
- The Koala compiler no longer supports multiple instantiation. The **multi** clause is documented here for historic reasons only.
- The keyword **no\_lib** specifies that this subcomponent should not be treated as library, and thus overrules a possible **library** keyword in the definition of that subcomponent.
- The Koala compiler no longer supports libraries. The **library** clause is documented here for historic reasons only.
- The keywords **multi** and **no\_lib** relate only to the subcomponent of which the instance name immediately precedes the keyword(s).
- The keywords **multi** and **no\_lib** can be specified in any order, but each keyword may occur at most once.

#### A.4.8 Connects

A *Connects* section (see Figure 90) specifies the ‘net-list’ of the component.

See sections A.4.9 for cable, A.4.11 for switch, and A.4.13 for within clauses.

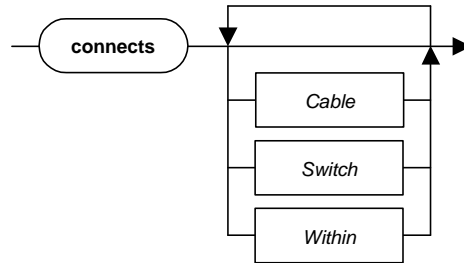


Figure 90. Connects

#### A.4.9 Cable

A *Cable* (see Figure 91) connects an interface to an interface, an interface to a module, or a module to an interface.

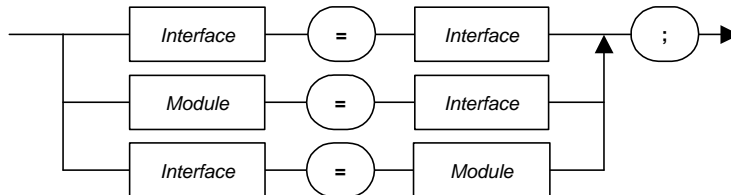


Figure 91. Cable

The following rules apply:

- *Interface* must refer to a valid interface instance in the component definition (see section A.4.10).
- **Definition:** the left-hand side interface (if present) is bound at its *tip* and is called the *tip interface* of this cable. The right-hand side interface (if present) is bound at its *base*, and is called the *base interface* of this cable.
- An interface instance can only be bound once at its tip, but can be bound zero or more times at its base.
- Provides interfaces of the component and requires interfaces of the subcomponents can only be bound at the tip in this component definition, requires interfaces of the component and provides interfaces of the subcomponents only at the base. Internal interfaces can be bound at both sides.
- If an interface is connected to an interface, then the type of the tip interface must be a subset of the type of the base interface (as defined in section A.3).

- A mandatory tip interface may only be connected to an optional base interface if Koala can determine at compile-time that the optional interface is present.
- If a module is connected to a base interface, then the module gets access to any function or data type (group) defined in that interface.
- If a module is connected to an *optional* base interface, then the module must check the *iPresent* function of that interface before using any function or parameter of the interface.
- Interface constants of optional interfaces can be used without check.
- If a mandatory interface is connected with its tip to a module, then this module must implement every function and every parameter of the interface (in C or in a Koala within clause, see section A.4.13).
- If an optional interface is connected with its tip to a module, then the module must implement the *iPresent* function of that interface (in C or in Koala). It only needs to implement the other functions of the interface if *iPresent* is not *false*.

#### A.4.10 Interface

An *Interface* (see Figure 92) refers to an interface instance of a component or of a subcomponent.

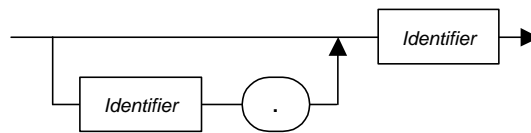


Figure 92. Interface

The following rules apply:

- If present, the first (optional) identifier names the subcomponent that provides or requires the interface.
- This subcomponent must be defined in the component (see section A.4.7).
- The second identifier names the actual interface instance.
- This interface must be defined on the subcomponent if present (as provides or requires interface) or in the component (as provides, requires or private interface).

#### A.4.11 Switch

A *Switch* (see Figure 93) allows conditional binding between sets of interfaces.

The following rules apply:

- The expression that follows the keyword **switch** may only refer to interfaces that are in scope: i.e. it may only use elements of provides, requires or private interfaces of the component, or of provides or requires interfaces of any of the subcomponents.

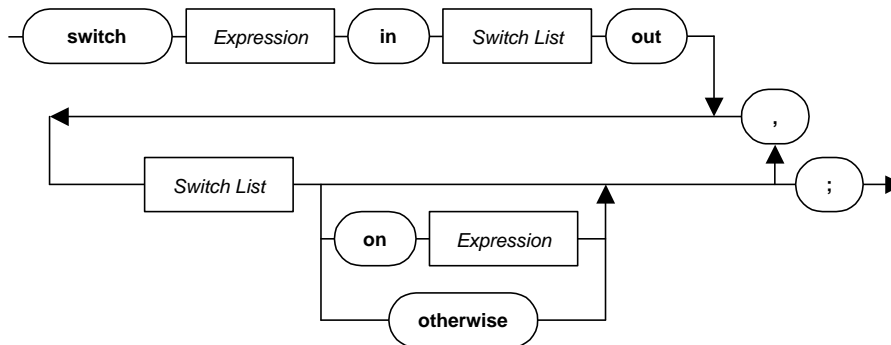


Figure 93. Switch

- The switch expression should evaluate to a numeric value. Note that Boolean expressions evaluate to 0 (**false**) or 1 (**true**) in Koala (see section A.4.16).
- The switch expression may not be a comma expression (see section A.4.18), i.e. have a comma as top level operator.
- All switch lists in a switch expression should be of equal length.
- Each interface of the **in** clause should match the corresponding (in position) interface of each **out** clause as if they were connected by a cable (see section A.4.9), unless Koala can decide at compile time that the connection will not be made.
- The switch connects the tip of each interface of the **in** clause to the base of each interface in the corresponding position of the selected switch list in the **out** clause.
- An **out** switch list is selected if its value corresponds to the value of the switch expression, or if it is specified as **otherwise** and no other **out** list has a value that matches the expression.
- The **otherwise** clause may only be attached to the last switch list in the **out** clause.
- An **out** switch list that has no **on** clause and no **otherwise** clause is called an implicit list. The first implicit list is assigned the value 0. Subsequent implicit lists are assigned the value of the previous implicit list plus 1. Note that explicit values (of **on** clauses) are not taken into account!

#### A.4.12 Switch List

A *Switch List* (see Figure 94) is a list of references to interface instances.

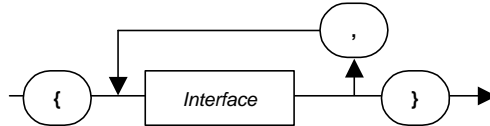


Figure 94. Switch List

The following rules apply:

- Each *Interface* must refer to a valid interface instance (see A.4.10).

#### A.4.13 Within

A *Within* clause (see Figure 95) allows to implement functions in Koala.

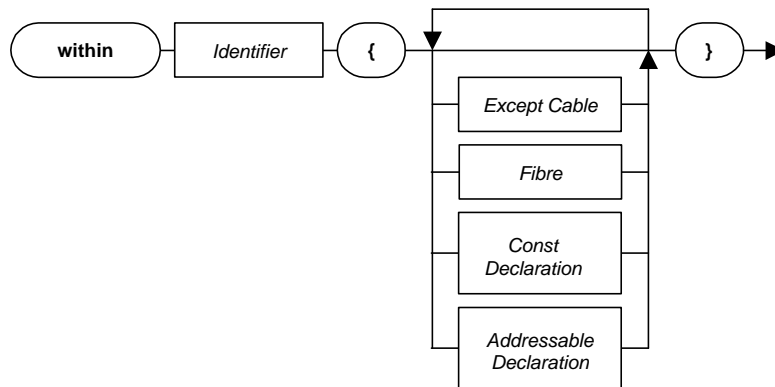


Figure 95. Within

The following rules apply:

- The identifier that follows the keyword **within** must refer to a module declared in the component as described in section A.4.6.
- The module declaration must precede the *within* clause for that module.<sup>19</sup>
- For each module, there must be at most one *within* clause.
- Modules for which no *within* clause exist are assumed to be completely implemented in C.

<sup>19</sup> The current Koala compiler does not enforce this rule, but future versions may.

- Modules for which a within clause exists may have any number of functions or parameters defined in Koala. Functions or parameters not defined in the within clause are assumed to be implemented in C.

#### A.4.14 Except Cable

An *Except Cable* (see Figure 96) implements an interface in terms of another interface with the exception of a specified set of functions or parameters.

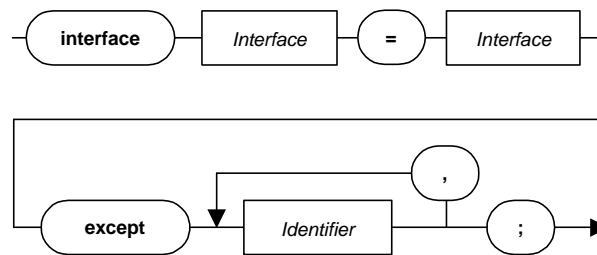


Figure 96. Except Cable

The following rules apply:

- An *Except Cable* is a *Cable* so all rules that apply to cables also apply here.
- The left hand side *Interface* must be bound with the tip to the module.
- The right hand side *Interface* must be bound with the base to the module.
- At most one *Except Cable* may be defined for a particular interface bound with the tip to the module.
- The *Identifiers* specify the functions that are excluded from this binding. These functions must be implemented separately in the within clause or in C.

#### A.4.15 Fibre

A *Fibre* (see Figure 97) allows to bind a function to a Koala expression.

The following rules apply:

- The *Interface* must be bound with its tip to the module.
- The *Identifier* refers to an element of the interface. This element must exist in the corresponding interface definition.
- An interface element can only be implemented once. If it is not implemented in a within clause, it must be implemented in C.
- For an interface parameter, the parameter clause (shaded in Figure 97) must be absent.

- For a function, the parameter clause must have the same length as the parameter list of the function in the corresponding interface type.

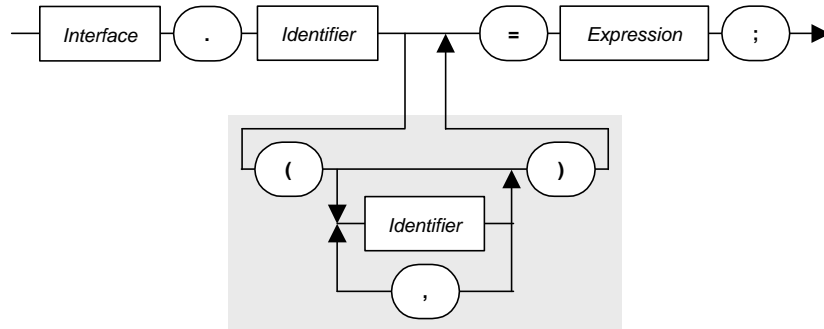


Figure 97. Fibre

- For a function, the parameter clause may also be absent if the expression consists of a reference to a single function with the same parameter list but specified without parameters (a so called *direct function binding*).
- The parameters in the parameter list must have distinct names. However, these names need not be the same as the parameter names in the interface definition.
- The expression may contain references to interfaces if they are connected to the module (both tip and base are allowed).
- The expression may contain identifiers if they refer to the parameters in the parameter clause.

#### A.4.16 Const Declaration

The *Const Declaration* clause (see Figure 98) allows to specify that certain functions or parameters of interfaces must be constant, i.e. Koala must be able to calculate the value.

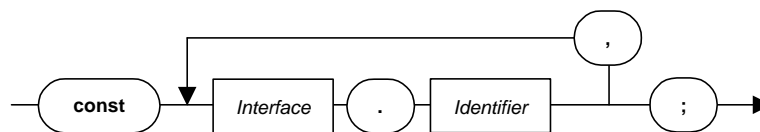


Figure 98. Const Declaration

The following rules apply:

- The *identifier* refers to a function or parameter of the interface. This function or parameter must exist in the corresponding interface definition.

#### A.4.17 Addressable Declaration

The *Addressable Declaration* clause (see Figure 99) allows to specify that certain functions of an interface must be addressable, i.e. it must be possible in the module to take the address of the function.

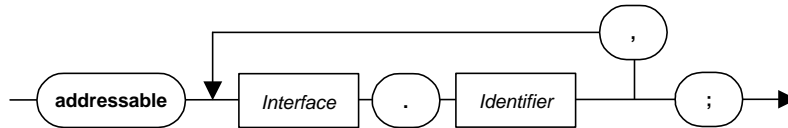


Figure 99. Addressable Declaration

The following remarks apply:

- The *Interface* must be bound with its base to the module.
- The *identifier* refers to a function or parameter of the interface. This function or parameter must exist in the corresponding interface definition.
- Koala makes sure that the interface element is indeed *addressable*. If necessary, Koala generates a function for this purpose.

#### A.4.18 Expression

An *Expression* (see Figure 100) calculates a value given an operator and one or more operand expressions.

The following rules apply:

- All operators have the same meaning as in the C language.
- All operators have the same priority as in the C language. Figure 100 can be read to show these priorities from low (top) to high (bottom). Operators that are grouped together in Figure 100 share the same priority.
- All operators have the same associativity as in the C language. For most of the operators,  $\alpha \circ \beta \circ \gamma$  is parsed as  $\alpha \circ (\beta \circ \gamma)$ . Use parentheses when in doubt!
- As in C, Boolean expressions (such as ‘!=’ and ‘<’) result in 0 for **false** and 1 for **true**.
- The *const* operator is not the same as const in C. It returns **true** if Koala can determine the value of the operand expression at compile-time, and **false** otherwise.
- Although the Koala compiler recognizes ‘U’ and ‘L’ suffices on numbers, *all* arithmetic is currently performed using signed integers of 32 bits.

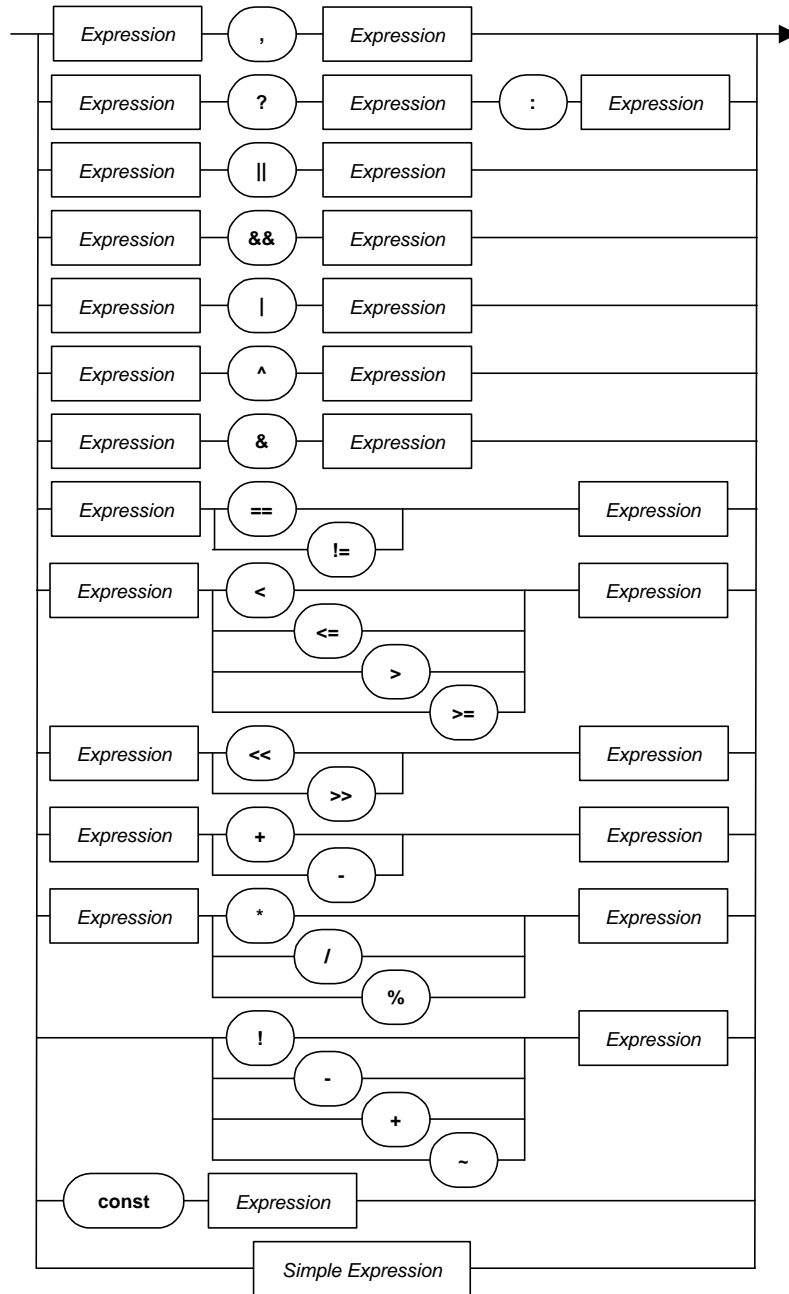


Figure 100. Expression

- Definition:** An expression that contains no inline expression is called *native*. An expression that does contain an inline expression is called *mixed* or *non-native*.

### A.4.19 Simple Expression

The syntax for a *Simple Expression* is shown in Figure 101.

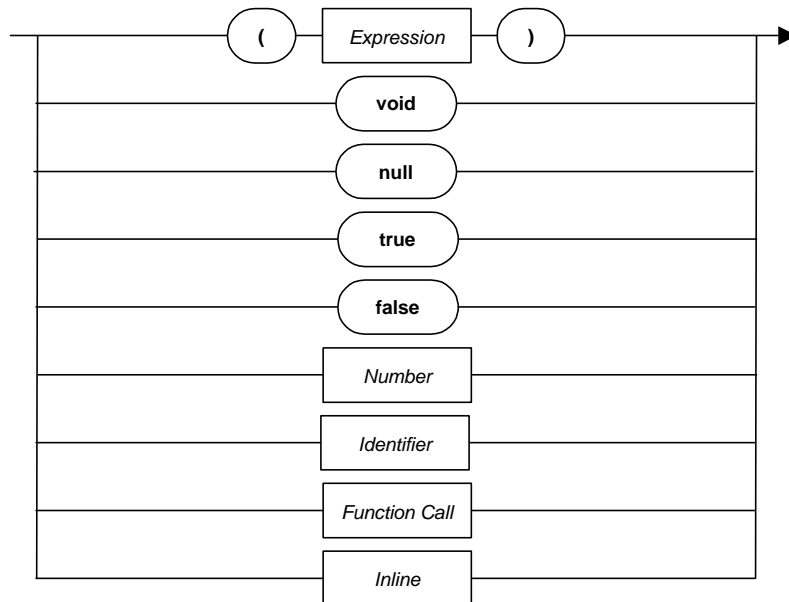


Figure 101. Simple Expression

The following rules apply:

- Parentheses can be used to change the order of evaluation.
- The keyword **void** denotes an empty expression. It can be used as body of void functions.
- The keyword **null** denotes an empty expression. It can be used as body of functions returning a (non-void) result.
- The value of **true** is 1. The value of **false** is 0.
- A *number* stands for itself.
- An *identifier* can only be used if the expression occurs in the right hand side of a function binding (see section A.4.15) or in the expression assigned to a function in an interface definition (see section A.3.3). The identifier must then refer to a formal parameter of the function.

### A.4.20 Function Call

A *Function Call* (see Figure 102) invokes an element of an interface.

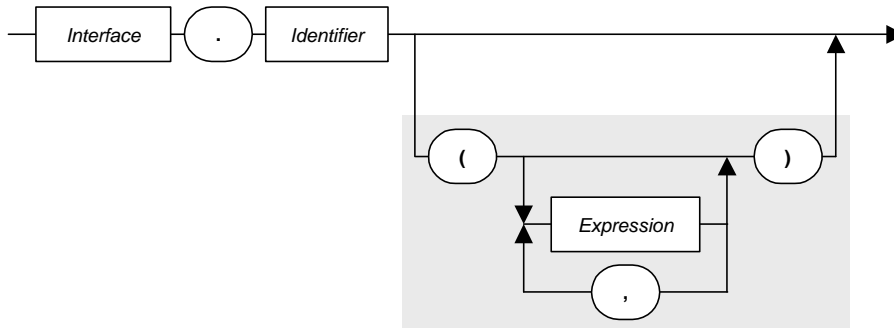


Figure 102. Function Call

## Remarks:

- *Interface* specifies the interface instance or definition of which the element is called.
- If the function call is part of an expression in an *interface* definition, then it can only refer to members of the same interface, and the interface must consist of the name of the interface definition only.
- If the function call is part of an expression in a *component* definition, then the interface must be valid as defined in section A.4.10, and it must be in scope as defined below.
- If the expression is part of a *within* clause (section A.4.13), the interface must be connected to the corresponding module (see section A.4.9) with the tip or with the base.
- If the expression is part of a *switch* (section A.4.11), then any interface of the component or of its subcomponents can be used.
- The *Identifier* specifies the element of the interface to be invoked.
- This element must be defined in the corresponding interface type.
- A parameter list (shaded in Figure 102) is not allowed for the invocation of an interface parameter or interface constant.
- A parameter list is mandatory for the invocation of a function in an interface.
- In that case, there must be one expression for each formal parameter.
- In a direct function binding (see section A.4.15), the parameter list must be omitted for a function 'call'.

**A.4.21 Inline Expression**

An *inline* expression (see Figure 103) allows to embed literal C texts in Koala.



Koala does not contain a specific data type definition language. Instead, data types are specified in C header files in the traditional way. To inform Koala which data types are defined in which header files, three pragmas are defined.

### A.5.1 Koala Type

A *Koala Type* clause (see Figure 104) declares that the header file contains a C definition for a data type.



Figure 104. Koala Type

The following remarks apply:

- The *Identifier* specifies the name of the data type being defined in this file. This name can be used in **using** or **uses** clauses (see sections A.3.2, A.4.4, A.4.21 and A.5.3).
- This name must be globally unique: it may not be equal to the name of any other data type, data type group, component type or interface type in the repository.

### A.5.2 Koala Group

A *Koala Group* clause (see Figure 105) declares that the header file defines a group of data types.



Figure 105. Koala Group

Remarks:

- The *Identifier* specifies the name for the data type group. This name can be used in **using** or **uses** clauses (see sections A.3.2, A.4.4, A.4.21 and A.5.3).
- This name may not be equal to the name of any other data type, data type group, component type or interface type in the repository.

### A.5.3 Koala Using

A *Koala Using* clause (see Figure 106) specifies that the data type definitions use data types defined in other header files.

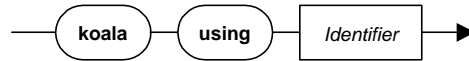


Figure 106. Koala Using

Remarks:

- The *Identifier* specifies the name of the data type or data type group being used in the header file.
- This identifier must refer to a data type or data type group defined somewhere in the repository.
- It is allowed to have multiple using clauses for the same data type or group in a file.

## A.6 Naming Conventions

In this section, we describe naming conventions that are recommended (*should, may*) or required (*shall, must*).

### A.6.1 Interface Definitions

The following conventions apply:

- An interface definition should have a name that is WordCased and prefixed with a capital 'I'. Example: *ITvSearchTuner*.
- An interface definition may specify (in comment) a prefix that can be used as preferred interface instance name. Example: *tun*.
- An interface definition must be stored in a file with extension '.id'.
- The name of this file should be equal the name of the interface definition.
- Storing multiple interface definitions in a single file is allowed though not preferred.
- Interface definitions can be stored anywhere in the repository.<sup>20</sup>

### A.6.2 Component Definitions

The following conventions apply:

- A component definition should have a name that is WordCased and prefixed with a capital 'C'. Example: *CTvSearchTuner*.

---

<sup>20</sup> MG-R poses extra restrictions on the location of Koala definitions in the directory structure.

- A component definition should specify a prefix (section A.4.2) that is lower case, does not contain any ‘\_’ characters, starts with a letter and is preferably 5-7 characters long. Example: *tvstun*.
- A component definition must be stored in a file with extension ‘.cd’.
- The name of this file should be equal to the long name of the component.
- Storing multiple component definitions in a single file is allowed but strongly discouraged.
- A component definition must be stored in a directory. All source files (.c and .h) found in that directory are considered part of the component.
- The name of this directory should be equal to the component short name.
- If multiple component definitions are stored in the same directory (not recommended!), then if any of the components is present in the product, all files in that source directory will be compiled (but only once).

### A.6.3 Data Type Definitions

The following conventions apply:

- Data type definitions are C definitions augmented with Koala pragmas (see section A.2).
- Data type definitions reside in normal header files, but these files should have extension ‘.dd’ rather than ‘.h’.
- Data type definitions may be stored anywhere in the repository.

### A.6.4 C and Header Files

The following conventions apply:

- Each C and header file should have a globally unique name, preferably one that starts with the component prefix followed by a ‘\_’ character.
- For each module that is not completely implemented in a within clause (see section A.4.13) there should be a C file with as name the component prefix followed by an ‘\_’ followed by the module name.<sup>21</sup>
- This C file must implement all functions of all interfaces connected with the tip to the module that have not been specified in a within clause.

---

<sup>21</sup> Actually, a module specifies a header file to be generated by Koala. This header file can be included in any C file of that component. But we strongly recommend to have one hand-written C file for each module.

- The C file must include the header file generated for the module. It may include other header files but only if they are located in the same directory.
- The C file may use any function, parameter or constant of any interface connected with the base or tip to the module.

### A.6.5 Implementing Functions or Parameters

The following conventions must be used when implementing functions or parameters in C:

- A function named 'f' of a provides interface 'p' or private interface 'i' of the component, where 'p' or 'i' is connected with the tip to the module, must be implemented as a C function named 'p\_f' respectively 'i\_f'.
- A function named 'f' of a requires interface 'r' of a sub-component with instance name 's', where 'r' is connected with the tip to the module, must be implemented as a C function named 's\_r\_f'.
- A static function may have an arbitrary name.
- A function with external linkage that is not implementing an interface function should have a name that starts with the component prefix followed by a single '\_'.
- A parameter can be implemented as a function, using the same naming conventions as described above, or (preferably) in a Koala inline expression.

### A.6.6 Using Functions, Parameters, Constants and Types

The following conventions apply when using functions, parameters or constants of interfaces connected to a module:

- A function, parameter or constant named 'f' of a provides, requires or internal interface named 'i' of the component containing the module, where 'i' is connected with the base or tip to the module, may be invoked as 'i\_f'.
- A function, parameter or constant named 'f' of a provides or requires interface named 'i' of a sub-component with instance name 's', where 'i' is connected with the base or tip to the module, may be used as 's\_i\_f'.
- Before calling any function or parameter in an optional interface, the caller should check whether the *iPresent* function of that interface yields **true**.
- Unless otherwise specified, once *iPresent* has returned true for a particular interface, it should continue to return true for that interface during the life time of the system.
- Constants of an optional interface can be used without checking *iPresent* first.

- Sometimes, a module assumes an interface element to be constant, e.g. when using it in an initialization expression of a static variable, as size of a static array, or in a case of a switch statement. If the element has not been declared constant in the interface definition, then this must be explicitly specified with a `const` declaration in the `within` clause for the module (see section A.4.16). Use of a `const` declaration in a `within` clause is not recommended!
- A module can inspect whether functions or parameters that have not been specified as constant still are constant in specific situations. The preprocessor macro `<f>_CONSTANT` is defined only if `<f>` evaluates to a constant.
- If the caller wants to take the address of a function of an interface, this must be explicitly specified with an addressable declaration (see section A.4.17) in the `within` clause for the module.
- A module can inspect whether an interface ‘`i`’ that is connected to it with the tip is actually connected at the outside. The preprocessor macro `i_CONNECTED` provides this information.<sup>22</sup>
- A module gets access to the definitions of all types mentioned in the definitions of all interfaces connected to the module, plus all types mentioned in the `uses` clause of the component that contains the module, plus all types mentioned in `using` clauses of inline expressions in function bindings for that module.
- A module can use the `PREFIX` macro to obtain the short name of the component.

## A.7 Const-Free Semantics

In this section, we define the semantics of the Koala language by providing a simple non-optimizing code generation scheme (also known as Koala0). The following restrictions hold:

- We implement single instantiation only (see section A.4.2).
- We do not do any (partial) evaluation of Koala expressions. Every expression is turned directly into either preprocessor or C code.
- We ignore the keyword `const` in Koala (as introduced in sections A.3.3, A.4.16 and A.4.18), which means that we assume that interface elements other than interface constants are never used at locations where C assumes constants.

---

<sup>22</sup> Although in principle `CONNECTED` should be equivalent to `iPresent`, Koala currently uses a much simpler algorithm to establish whether interfaces are connected.

- We generate code for every component in the repository independently, i.e. without looking into other components (but with using the type information of provides and requires interfaces of subcomponents).
- At occasions, we may make more definitions visible than the language describes.

The resulting code for a component can be compiled independently from other components. An application can be built by linking the top-level component and all components recursively included.

### A.7.1 Generated Files

For each component, a C file will be generated (see section A.4.2 for the name of this file) that contains:

- *#include* statements for all relevant data types (see section A.7.2).
- *#define* statements for all interface elements that have an expression assigned in the definitions of interfaces of the component or its subcomponents (see section A.7.5).
- Call-through functions for interface-interface bindings (see section A.7.6).
- Select functions for switches (see section A.7.7).

For each module, a header file will be generated (see section A.4.6) that contains:

- *#include* statements for all relevant data types (as explained in section A.7.2).
- *#define* statements for all interface elements with an expression assigned in the definitions of interfaces of the component or its subcomponents (see section A.7.5).
- Function definitions for all interface elements that are assigned an expression in a within clause (see section A.7.8).<sup>23</sup>
- *#define* statements and extern declarations for all other interface elements (see section A.7.9).
- A definition of *i\_CONNECTED* for every interface ‘i’ connected with the tip to the module:

```
#define i_CONNECTED 1
```

---

<sup>23</sup> Note that the generated header file contains C code for functions implemented in within clauses! This is no problem as long as the header file is only included in one C file. An alternative is to generate the code in the component C file

### A.7.2 Data Type Dependencies

The C file generated for the component includes those '.dd' files that contain data types or data type groups that occur:

- In interfaces of the component (provides, requires or internal) or of its subcomponents (provides or requires).
- In *uses* statements of the definitions of interfaces mentioned above.
- In *using* statements of expressions in the definitions of interfaces mentioned above.
- In *uses* statements of the component.
- In *using* statements of '.dd' files already included.

The module header file contains includes for those '.dd' files that contain data types or data type groups that occur:

- In interfaces of the component (provides, requires, internal) or of its subcomponents (provides, requires) that are bound to the module.
- In *uses* statements of the definitions of the interfaces mentioned above.
- In *using* statements of expressions in the definitions of the interfaces mentioned above.
- In *using* clauses of expressions in fibers in the within clause for the module.
- In *uses* statements of the component.
- In *using* statements of '.dd' files already included.

All includes are generated in the right order<sup>24</sup> and without any duplications<sup>25</sup>.

### A.7.3 Physical Names

A function 'f' of an interface instance 'i' of a component with prefix 'c' will become a C function with name 'c\_\_i\_f', regardless of the role of the interface (provides, requires, internal).

A parameter 'p' of an interface instance 'i' of a component with prefix 'c' will also become a C function with name 'c\_\_i\_p', regardless of the role of the interface. Note that 'p' is referred to *without* parentheses, both in C and in Koala!

---

<sup>24</sup> If A uses B then B is included before A.

<sup>25</sup> An easy way to avoid duplications is by surrounding each include with an *#ifdef/#endif* pair. Note that a simple (be it less restrictive) solution to generating the module header file is to copy the information written to the component C file (but using clauses of expressions in fibers must be added then).

Remarks:

- If a function or parameter is assigned an expression in the interface definition, then it will only have a logical name.
- Functions and parameters in provides and internal interfaces are defined in the context of the component that owns them. Functions and parameters of requires interfaces are defined in the context of the (compound) component that instantiates the component that owns them. Note that there can be more than one such component in the repository!

#### A.7.4 IPresent

For each mandatory interface ‘i’ (provides, requires or internal) in a component with prefix ‘c’, the following function is generated in the component C file<sup>26</sup>:

```
int c__i_iPresent(void)
{
    return 1;
}
```

For each optional provides or internal interface of a component or requires interface of a subcomponent that is *not* connected, the following function is generated in the component C file:

```
int c__i_iPresent(void)
{
    return 0;
}
```

Remarks:

- ‘c’ is the prefix of the component that owns interface ‘i’.

In sections A.7.6, A.7.7, A.7.8 and A.7.9, optional interfaces are considered to have an extra function *iPresent*.

#### A.7.5 Interface Expressions

An interface *constant* ‘c’ of an interface ‘i’ of type ‘I’ is translated to a #define as follows:

```
#define i_c <expression>
#define i_c_CONSTANT <expression>27
```

Remarks:

---

<sup>26</sup> One could argue that *iPresent* for a mandatory requires interface of a component C should be implemented in the compound component that instantiates C.

<sup>27</sup> In the ‘real’ Koala compiler, the expression for ‘i\_c’ may contain C type casts, while the expression for ‘i\_c\_CONSTANT’ may only have constructs that are known to the C preprocessor.

- The C *<expression>* is generated from the corresponding Koala expression using the scheme described in section A.7.10, with one exception:
- References to elements ‘x’ of the same interface are translated from ‘I.x’ to ‘i\_x’, where ‘I’ is the type and ‘i’ the instance name of the interface.

An interface *function* ‘f’ of an interface ‘i’ of type ‘I’ is translated to a #define as follows:

```
#define i_f(<params>) <expression>
```

Remarks:

- The parameter list consists of parameter names only (no types).
- The expression is translated according to section A.7.10, with the same exception as mentioned above.
- There is no `_CONSTANT` macro being defined.

### A.7.6 Interface-Interface Binding

For each function ‘f’ in the left hand side of an interface-interface binding, a C function will be generated in the component C file of the form:

```
<type> <c>__<lhs>_f ( <parameter declarations> )
{
    return <cc>_<rhs>_f( <parameters> );
}
```

Remarks:

- If the *<type>* of the function is void, then the *return* keyword is omitted.
- The *parameter declarations* may be empty.
- *<c>* is the prefix of the component that owns the interface *<lhs>*.
- *<cc>* is the prefix of the component that owns the interface *<rhs>*.

For each parameter ‘p’ in the left hand side of an interface-interface binding, a C function will be generated of the form:

```
<type> <c>__<lhs>_p ( void )
{
    return <cc>__<rhs>_p ( );
}
```

### A.7.7 Switches

For each function ‘f’ in an interface ‘i’ that is input to a switch, a function will be generated in the C file of the component of the following form:

```

<type> <c>__i_f ( <parameter declarations> )
{
    int e = <switch expression>;
    if ( e == <value of first output choice>
        return <ccl>__<ocl>_f ( <parameters> );
    else ...
}

```

Remarks:

- *<c>* is the prefix of the component to which ‘i’ belongs. This can either be the compound component, or one of its subcomponents.
- The *parameter declarations* are copied from the interface definition.
- The *switch expression* is only evaluated once.
- For *void* functions, the *return* keyword is omitted.
- *<ocl>* is the interface to which this output choice is bound for the particular value of the expression.
- *<ccl>* is the component that owns *<ocl>*.
- The *parameters* contain the names of the formal parameters of the function.

For each parameter ‘p’ in an interface ‘i’ that is input to a switch, the generated code is similar:

```

<type> <c>__i_p(void)
{
    int e = <switch expression>;
    if ( e == <value of first output choice>
        return <ccl>__<ocl>_p ();
    else ...
}

```

### A.7.8 Fibres

For a function ‘f’ of an interface ‘i’ that is implemented in a within clause in Koala, a function of the following form is generated:

```

#define i_f c__i_f
<type> c__i_f(<param list>)
{
    return <expression>;
}

```

Remarks:

- ‘c’ is the prefix of the component that owns the interface ‘i’.
- *<type>* may be void, in which case the keyword *return* is omitted.
- The C *<expression>* is generated from the Koala expression following the translation scheme described in section A.7.10.

For a parameter ‘p’ of an interface ‘i’ that is implemented in a within clause in Koala, a function of the following form is generated:

```
#define i_p c__i_p()
<type> c__i_p()
{
    return <expression>;
}
```

### A.7.9 Interface-Module Binding

For each function ‘f’ of an interface ‘i’, where ‘i’ is connected to a module ‘m’ and ‘f’ is not assigned an expression in the interface definition of ‘i’ nor in a within clause for the module ‘m’, the following code is generated in the header file of module ‘m’:

```
#define i_f c__i_f
extern <type> c__i_f(< parameter declarations>);
```

Remarks:

- ‘c’ is the prefix of the component that owns ‘i’.
- The function may have a void type.
- The parameter declarations are copied from the interface definition.
- If the interface ‘i’ is optional, and *iPresent* is not implemented in a within clause, then the code above is generated with  $f = iPresent$ .

For each parameter ‘p’ satisfying the same conditions, the following code is generated:

```
#define i_p c__i_p()
extern <type> c__i_p(void);
```

If the interface ‘i’ is owned by a subcomponent with instance name ‘s’ and prefix ‘cc’, then the following code is generated:

```
#define s_i_f cc__i_f
extern <type> cc__i_f(< parameter declarations>);
```

Similarly for a parameter:

```
#define s_i_p cc__i_p()
extern <type> cc__i_p(void);
```

### A.7.10 Expressions

Since the Koala expression language is derived from the C expression language, translation will be straightforward. We only discuss the non-trivial part here.

- *const* <expression> will be translated to 0 (for *false*).
- A *void* expression will be translated to *while (0) { }*.
- An *inline* expression will appear as is. All string constants in the *using* clause will be inserted in the generated code before the definition in which

the current expression appears. Function references are already declared in the context. Data type and data type group references of the using clause have been dealt with in section A.7.2.

- References to (other) elements of (other) interfaces will be converted from dot notation (i.f) to underscore notation (i\_f).

## **A.8 Concluding Remarks**

Reachability is part of the optimization facilities of the Koala compiler, and is not discussed in this document.