# Syndex : User manual

Maxence Guesdon, Cécile Stentzel, Meriem Zidouni

January 16, 2014

# Contents

# Chapter 1

# Overview

## 1.1 The AAA methodology

SynDEx is based on the *AAA methodology* (*cf.* [1]).

A SynDEx application is made of:

- *algorithm graphs* (definitions of operations that the application may execute),
- *architecture graphs* (definitions of multicomponents: set of interconnected processors and specific integrated circuits).

Performing an *adequation* means to execute heuristics, seeking for an optimized *implementation* of a given algorithm onto a given architecture.

An implementation consists in:

- *distributing* the algorithm onto the architecture (allocate parts of algorithm onto components),
- *scheduling* the algorithm onto the architecture (give a total order for the operations *distributed* onto a component).

## 1.2 SynDEx distributions

SynDEx runs under Linux, Windows, and Mac OS X platforms but is currently delivered only for Linux 64 bits platforms.

SynDEx is written in *Objective Caml*. The Graphical User Interface uses the *Gtk* library through the *LablGtk* bindings.

# Chapter 2

# Using the SynDEx tools

The SynDEx distribution comes with various tools :

- **syndex-sdxconv** translates SynDEx v7 files to SynDEx 8,
- **syndex**, the main command-line application, allowing to load all data to perform adequation and generate code,
- **syndex-durations** is a command-line tool to handle duration files,
- **syndex-adeq-gui** is used to display the result of adequation heuristics.

## 2.1 syndex-sdxconv

**Synopsis : syndex-sdxconv [options] <syndex v7 .sdx file>**

This tool is used to convert SynDEx v7 **.sdx** files to the SynDEx 8 syntaxes. The tool reads the given SynDEx v7 **.sdx** file and outputs algorithms, architectures and/or durations in SynDEx 8 syntax.

Options are :

| | |
|---|---|
| **--version** | Print version number and exit |
| **-I dir** | Add **dir** to the list of include directories (where to search files) |
| **--algo** | Dump algorithms on standard output (default action) |
| **--arch name** | Dump architecture definition **name** (with types) on standard output |
| **--dur** | Dump durations on standard output |
| **--cons** | Dump constraints on standard output |
| **--split prefix** | Dump algorithms, architectures, durations, and constraints into files with prefix **prefix** |

The format of algorithm files is described in section 3. The format of architecture files is described in section 4. The format of duration files is described in section 5. The format of constraints files is described in section 6.

Splitting the application creates several files for the architectures : one module file for all the declarations of types and one module file for each architecture definition.

Example of a **basic.arc** file containing the declarations of types :

```
medium type mediumsammultipoint : SAMMP
[...]
operator type uin {
  gate x : mediumsampointtopoint;
  gate y : mediumsammultipoint;
  gate z : mediumram;
}
[...]
```

Example of a **basic_ArchiSamMultiPoint.arc** file containing an architecture definition :

```
medium medium_sammp : Basic.mediumsammultipoint

operator u1 : Basic.uin {
  y -> medium_sammp;
}
[...]
operator u3 : Basic.uout {
  y -> medium_sammp;
}
```

## 2.2 syndex

**Synopsis : syndex [options]**

Options are of two kinds : general options and action options.
The general options are :

| | |
|---|---|
| **--version** | Print version number and exit |
| **-I dir** | Add **dir** to the list of include directories (where to search files) |

The action options define actions which will be performed by **syndex** in the given order. Here is the list of these actions :

| | |
|---|---|
| **--algo file.alg** | Read algorithm from **file.alg** |
| **--print-algo** | Print algorithm on standard output |
| **--dot-algo file-alg.dot** | Dump algorithm graph into **file-alg.dot** using the graphviz format |
| **--flatten name** | Flatten algorithm from the given definition name |
| **--dot-flat file-flat.dot** | Dump flat algo graph into **file-flat.dot** using the graphviz format |
| **--arch file.arc** | Read architecture from **file.arc** |
| **--print-arch** | Print architecture on standard output |
| **--dot-arch file-arc.dot** | Dump architecture graph into **file-arc.dot** using the graphviz format |
| **--durations file.fd** | Load operation durations from rules in **file.fd** |
| **--com-durations file.cd** | Load communication durations from rules in **file.cd** |
| **--constraints file.cts** | Load constraints from rules in **file.cts** |
| **--adeq-greedy** | Compute adequation with greedy heuristic |
| **--adeq-greedy-mp** | Compute multi-periodic adequation with greedy heuristic |
| **--dot-adeq file-adeq.dot** | Dump adequation result into **file-adeq.dot** using the graphviz format |
| **--dump-adeq file-adeq.adq** | Dump adequation result into **file-adeq.adq** |
| **--display-adeq** | Display adequation |

### Algorithms

The **syndex** tool keeps a current algorithm. When launching **syndex**, this algorithm is empty. The **--algo** option is used to add the algorithm defined in a file to the current algorithm. This option can be used several times to add several algorithms to the current algorithm. Of course, if two loaded algorithms contain two definitions with the same name, an error will be reported when loading the second algorithm.

The format of algorithm files is described in section 3.

### Architectures

The **syndex** tool keeps a current architecture. When launching **syndex**, this architecture is empty. The **--arch** option is used to add architecture declarations (declarations of types, media or operators) from a file to the current architecture. This option can be used several times to add declarations from several files to the current architecture. Of course, if two loaded files contain two declarations with the same name, an error will be reported when loading the second one.

The format of architecture files is described in section 4.

### Flattening

Flattening is performed with the **--flatten** option, giving a definition name to start the flattening from. This definition must be given arguments if it needs. Here are some examples :

```
# syndex ...  --flatten main ...  # main requires no argument
# syndex ...  --flatten "main(1)" ...  # main requires one argument
# syndex ...  --flatten "main(30, 12, 10)" ...  # main requires three arguments
```

Using adequation heuristics is possible only when the algorithm has been flattened. Then, the adequation is performed on the current flattened algorithm and the current architecture. Note that loading an additional algorithm after the **--flatten** option destroys the current flattened algorithm to avoid confusion.

### Adequation parameters and heuristics

Operation durations, communication durations and placement constraints are respectively loaded with options **--durations**, **--com-durations** and **--constraints**.

The order of options matters: if an adequation was computed with an option before one of these options, it is destroyed to avoid confusion. A message like **No adequation available** might appear.

The format of duration files is described in section 5 and the format of constraint files is described in section 6.

An adequation heuristic is used with one the following options: **--adeq-greedy**, **--adeq-greedy-mp**. An error will be reported if no flattened graph is available yet.

The result of the heuristic can be stored in a (binary) file using the **--dump-adeq** option.

So it is possible to perform several adequations with several parameters sequentially with one **syndex** command. Here is an example of command performing twice the greedy adequation heuristic, with two different sets of operation durations :

```
# syndex --algo my-algo.alg --flatten main --arch my-arch.arc \
--constraints myconstraints.cts --com-durations mycomdurs.cd \
--durations mydurs1.fd --adeq-greedy --dump-adeq adeq1.adq \
--durations mydurs2.fd --adeq-greedy --dump-adeq adeq2.adq
```

Binary dumps of adequation results can be viewed with the **syndex-adeq-gui** tool (see section 2.4) or with the **--display-adeq** option. Note that you may have to modify your PATH to help **syndex** finding the **syndex-adeq-gui** tool.

## 2.3 syndex-durations

**Synopsis : syndex-durations [options] <files>**

This tool can be used to handle (operation and communication) duration files (see section 5). The given duration files are loaded and the result of the grouping/expanding is written on standard output.

Options are :

| **--version** | Print version number and exit |
|---|---|
| **--by-1** | Group rules by factorizing elements of first field (default) |
| **--by-2** | Group rules by factorizing elements of second field |
| **--expand** | Expand all rules |

## 2.4 syndex-adeq-gui

**Synopsis : syndex-adeq-gui <file>**

This tool is used to graphically display the result of an adequation heuristic, stored in the file in argument. This file must be a binary dump produced by the **--dump-adeq** option of the **syndex** tool.

# Chapter 3

# Defining algorithms

Algorithms are defined in **.alg** files, using a specific syntax.

An algorithm file contains the declarations of definitions, one after another, in precedence order, that is, if definition $d_1$ contains a reference to definition $d_2$, then $d_2$ must appear before $d_1$ in the algorithm file.

Remember that algorithms are composed of :

- definitions of various kinds : function, sensor, actuator, delay, constant,
- references to definitions,
- ports of references and definitions.

Definitions have ports and contain references to other definitions. If a definition $d$ have ports, then the reference $r$ to $d$ automatically has the same ports as $d$. We say that $r$ references $d$ and each port of $r$ references a port of $d$.

A definition $d$ also have links, representing the data flow. There are various possibilities for links :

- links from an input port of $d$ to an output port of $d$,
- links from an input port of $d$ to an input port of a reference contained in $d$,
- links from an output port of a reference contained in $d$ to an output port of $d$,
- links from an output port of a reference $r_1$ to an input port of a reference $r_2$, with $r_1$ and $r_2$ contained in $d$.

Sensor and constants can only have output ports. Actuator can only have input ports. Delays can only have one input port and one output port.

References may have some attributes :

- a period, in case of a multi-periodic application,
- an abstract flag, indicating that the reference must be considered as a "leaf" when flattening the algorithm,
- repetition information, used to repeat references when flattening the algorithm; repetitions can be sequential or parallel.

Ports are defined by their direction (input or output), their type and their size.

At last, definitions can have parameters. A reference to a definition having $n$ parameters must have $n$ arguments.

## 3.1 Expressions

Expressions are used to express port sizes and repetition factors. The arguments to references are expressions, too. The parameters of a definition are identifiers which are bounds to the arguments of the reference when flattening the graph.

By now, the expressions are composed only of integer constants, variable names, binary operations ('+', '-', '*' and '/') and unary negation '-'. Parenthesis are used as usual to deambiguate expressions, and '*' and '/' have higher priority than '+' and '-'.

Here are some examples of valid expressions :

- 1
- 1 + 2
- $(1 + 2) * (- 3 - 1)$
- $(a + b) * 3 / 1$
- x

## 3.2 Definitions

A definition is describe using the following general syntax.

```
<definition kind> name <parameters> { <body> }
```

The definition kind is one of the following keywords : **actuator**, **constant**, **delay function**, **sensor**.

The name is an identifier composed of any of the characters **'a'..'z'**, **'A'..'Z'**, **'0'..'9'**, **'_'** and beginning with a lowercase letter (**'a'..'z'**).

If the definition has parameters, they are indicated between parentheses and separated by commas. Each parameter must be a valid simple identifier following the same lexical convention as definition names.

The syntax of the definition body depends on the kind of the definition.

### 3.2.1 Sensors and actuators

The body of **sensor** and **actuator** definitions is composed only of a list of port declarations (see below). Examples :

```
sensor input1 { int[3] data1 , char data2 }
actuator output1 (size) { int[size] data }
```

### 3.2.2 Constants

The body of **constant** definitions is composed only of a list of declarations of ports and values. Example :

```
constant cst { int o = 2, factor = 42 }
```

### 3.2.3 Delays

The body of delays are only composed of port declarations, with one input port and one output port. Example :

```
delay memory {
  in: int[2] v_in;
  out: int[2] v_out;
}
```

### 3.2.4 Functions

The body of a function is composed of port declarations, references declarations and edge declarations.

**Ports**

Functions can have any number of input and output ports. A port is defined by its name, its type and its size, with the following syntax :

```
<type> <optional size> <name> <optional extension attributes>
```

The type is any identifier but is used during flattening to check that types of connected ports are consistent. The optional size is a valid expression and must be indicated between square brackets **'['** and **']'**. If no size is given, the expression **1** is assumed. The name of a port must follow the same lexical conventions as definition names. Input (resp. output) ports are specified with the **in** (resp. **out**) keyword, as in :

```
function foo (a, b) {
  in: int bar, float[b] gee ;
  out: char[a+b] buz [x="16", y="478"];
  ...
}
```

Extension attributes of ports are similar to extension attributes of references which are explained below.

**References**

References are introduced with the construction

```
let <reference_name> = <qualified_definition_name> <optional
arguments> <optional attributes> ;
```

A reference name is a lowercase identifier following the same lexical convention as definition names. The qualified definition name is either a simple definition name or a name with the form **Module_name.definition_name** (see section 3.5 about modules).

If the referenced definition has parameters, arguments must be given after the definition name, between parentheses. These arguments are any valid expressions (see section 3.1).

A reference can have various attributes, indicated with a comma-separated list between square brackets '[' and ']'. There are two kinds of attributes : "predefined" attributes and "extension" attributes.

Predefined attributes allow setting some properties about the defined references. Here are the existing predefined attributes :

- **abstract** : when this attribute is specified, the reference will be considered as a final component (a 'leaf') of the algorithm during the flattening, which means that the flattening will not flatten the algorithm below (the definition referenced by) this reference,
- **period** : this attribute, followed by an integer, indicates the period of the reference, in multi-periodic applications,
- **parallel** and **sequence** : these attributes, followed by a repetition factor *e*, indicate that this reference must be repeated *e* times during the flattening. *e* is an expression. Repetitions are explained in section 3.4.

Here is an example of reference declarations :

```
function example (a, b) {
  ...
  let foo1 = compute1 (1, b) [ abstract, period 3 ] ;
  let foo2 = compute1 (a, 12) [ period 5, parallel (a + b) ] ;
  let foo3 = compute2 ;
  ...
}
```

Extension attributes are pairs (key, value) allowing to associate any string value to any key. This will be useful in the future to allow syndex extensions (others heuristics) to specify and use additional information about references. Extension attributes are given with the syntax **key="value"** in the attribute list. Here is an example :

```
function example (a, b) {
  ...
  let foo4 = compute1 (a, b) [ x="125", y="257", period 12 ] ;
  ...
}
```

These extension attributes are used for example by graphical editors to store the coordinates of references in graphical display of definitions.

### Edges

A definition contains edges between ports to indicate data flow, and edges between references to indicate a precedence order.

A data flow between two ports $p_1$ and $p_2$ is indicated with the syntax $p_1$ **->** $p_2$. A port of the current definition is given by its name, while a port of a reference is qualified by the reference name. The example code below creates the algorithm of figure 3.1.

```
function foo {
  in: int i;
  out: int o;
}

function bar {
  in: int v_in ;
  out: int v_out1, int v_out2;
  let r1 = foo ;
  let r2 = foo ;
  v_in -> r1.i ;
  v_in -> r2.i ;
  r1.o -> v_out1 ;
  r2.o -> v_out2 ;
  r1 --> r2 ;
}
```

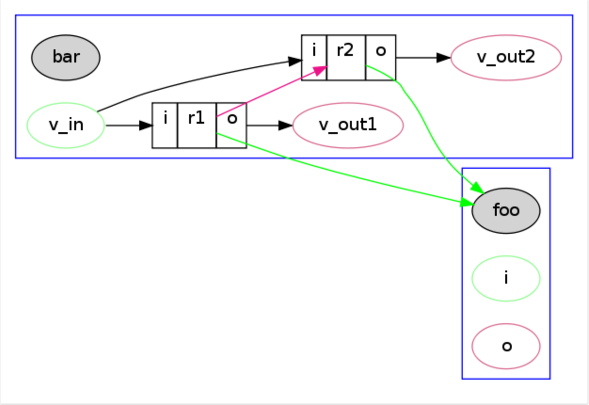

Figure 3.1: Algorithme with edges.

In the example above, a precedence edge from **r1** to **r2** (**r1 --> r2**) is also added to indicate that reference **r1** must be scheduled before the reference **r2**.

## 3.3 Conditioning

Conditionning is introduced by the following construction in the body of a function :

```
match <port name> with
n1 -> < reference and edges declarations >
| n2 -> < reference and edges declarations >
| ...
```

The ports of the definition must be declared before the conditioning structure. The **port name** must be one of the input port of the definition. **n1**, **n2**, . . . , must be integer constants (by now). For each case, references and edges can be added. One can add two references with the same name if they are in two different cases. Under each case, only the ports of the definition and the references under this case are known, so edges under each case can only concern ports of the definition and references under this case. Nested conditions are not allowed. To represent nested conditions, one must create one definition per condition.

The following code defines and used a conditioned function, and figure 3.2 shows the resulting algorithm :

```
function switch_1() {
  in: int i, int cond;
  out: int o;
  match cond with
  | 1 ->
      let r_mul = Int.arit_mul(1) ;
      cond -> r_mul.b ;
      i -> r_mul.a ;
      r_mul.o -> o ;
  | 2 ->
      let r_add = Int.arit_add(1) ;
      cond -> r_add.b;
      i -> r_add.a;
      r_add.o -> o;
}
```

```
function main() {
  let switch_1 = switch_1 ;
  let input_2 = Int.input(1) ;
  let input_1 = Int.input(1) ;
  let output = Int.output(1) ;
  switch_1.o -> output.i;
  input_2.o -> switch_1.cond;
  input_1.o -> switch_1.i;
}
```

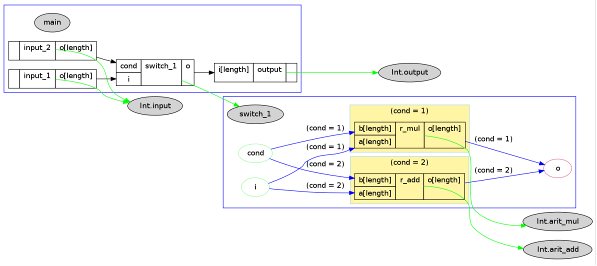

Figure 3.2: Algorithm with conditioning.

## 3.4 Repetitions

The specified algorithm may contain indications to repeat some references. It is a way to ease the definition of the algorithm. Indeed, these repetitions are not dynamic but static. Before the flattening of the algorithm, repetitions are handled to obtain an algorithm as if references had been manually repeated in the specification.

Two kinds of repetition exist : parallel or sequential.

### 3.4.1 Parallel repetition

The parallel repetition indicated on a reference consists in duplicating the reference and adding two additional operations :

- upstream of these references, an *Explode* operation allows to distribute, between the various instances of the repeated reference, the data provided by the ports producing the data consumed by the repeated reference,
- downstream, an *Implode* operation allows to collect the output data of these instances to send it to ports consuming the data of the repeated reference.

Figure 3.3 shows the flattened graph obtained from the following algorithm :

```
function f() {
  in: int[2] i1, int i2, int[3] i3 ;
  out: int[6] o1, int o2 ;
}

sensor capteur() { int[6] o1, int o2, int[3] o3 }
actuator actionneur() { int[6] i1, int[3] i2 }
```

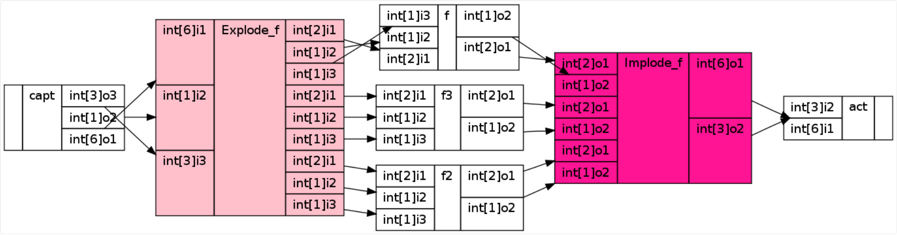

Figure 3.3: Example of parallel repetition.

```
function main() {
  let capt = capteur ;
  let act = actionneur ;
  let f = f [ parallel 3 (i2=, i3/, o2*, o1/) ] ;
  capt.o1 -> f.i1 ;
  capt.o2 -> f.i2 ;
  capt.o3 -> f.i3 ;
  f.o1 -> act.i1 ;
  f.o2 -> act.i2 ;
}
```

This example illustrates the three possible strategies to distribute the data in a parallel repetition :

- the port **i** of the reference **f** has no particular indication in the parallel repetition attribute, so it is given the "Mul" strategy, meaning that the producers of the data feeding this port must produce $N$ times more data to feed all repetitions ($N$ being the repetition factor, here $N = 3$). Since the size of port **i1** of **f** is 2, the port **o1** of **capteur** has its size set to $2 * N = 6$ (by the user) to that port sizes are consistent ; so the port **i1** of **Explode_f** has a size of 6 and its output ports **i1** has each a size of 2,
- the choice for port **i3** of **f** is symetrical : the producers of the data feeding **i3** don't have to produce more data, but we divide the port size of repetitions of **f** by $N$, which gives a size of 1, with consistent sizes for the ports **i3** of **Explode_f** ; this is the "Div" strategy,
- fort port **i2** of **f**, we choose to send the same data to all repetitions of **f**, so all input and output ports **i2** of **Explode_f** have the same size ; this is the "Equal" strategy.

The reasoning is similar for the **Implode_f** operation and the output ports of the repetitions of **f**, except that the "Equal" strategy is not allowed because the Implode operation cannot choose which data to transmit among all the received data (the repetitions will usually not output the same data).

Remark : The repetition does not modify the size of ports of data producers and consumers of **f**, bu controls are performed to check consistency between these sizes. The user is responsible for giving correct sizes.

### Répétition séquentielle

The sequential repetition consists in repeating a reference to create a *pipeline* with these repetitions. The data source for each input port of repetitions must be indicated, with two alternatives :

- If a link is given from an output port **o** to an input port **i** (with the syntax **i <- o**), this means that between two copies $r_1$ and $r_2$ in sequence in this order, the data from port **o** of $r_1$ will feed the port

**i** of $r_2$. The prot **i** of the first reference of the pipeline will consume the data of the producer of the original (repeated) reference in the algorithm and data or port **o** of the last reference of the pipeline will feed to the ports consuming the port **o** of the original (repeated) reference of the algorithm,

- If a distribution strategy is indicated for an input port, this strategy is used in an Explode operation created and feeding the corresponding ports, the samy way as for parallel repetitions (see above).

If no information is given about one input port, the "Mul" distribution strategy is used to feed this port in each copy of the repeated reference, from that data producer of the original algorithm.

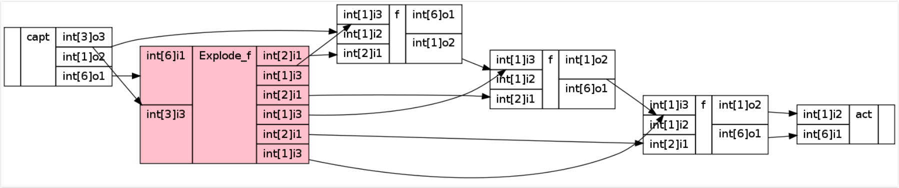

Figure 3.4: Example of sequential repetition.

Figure 3.4 show the flattened algorithm for the following algorithm :

```
function f() {
  in: int[2] i1, int i2, int[3] i3 ;
  out: int[6] o1, int o2 ;
}

sensor capteur() { int[6] o1, int o2, int[3] o3 }
actuator actionneur() { int[6] i1, int[1] i2 }

function main() {
  let capt = capteur ;
  let act = actionneur ;
  let f = f [ sequence 3 (i3/, o2->i2) ] ;
  capt.o1 -> f.i1 ;
  capt.o2 -> f.i2 ;
  capt.o3 -> f.i3 ;
  f.o1 -> act.i1 ;
  f.o2 -> act.i2 ;
}
```

## 3.5 Modules

A module system allows separate definition of pieces of algorithm. A module contains definitions and can be seen as a library. As soon as a file **foo.alg** contains one definition, another algorithm file can use the module **Foo** to use the definitions in **foo.alg**.

Refering to a module definition is done simply by prefixing the definition name with the module name, as in **Int.arit_add** which refers to definition **arit_add** of module **Int**.

When a module name appears, the corresponding algorithm file is looked up in the *include directories* (specified by **-I** options of the SynDEx tools). The first file named "foo.alg" (in a case-insensitive search) will be considered as the algorithm file for module **Foo**. This file is searched in the order in which include directories were specified (except that the file is always searched first in the current directory).

## 3.6 BNF

Here is the algorithm specification language represented in Backus-Naur form :

```
algo ::= def*

def ::= constant | sensor | actuator | delay | function

constant ::= 'constant' name '(' params ')' '{' port_values '}'

sensor ::= 'sensor' name '(' params ')' '{' ports '}'

actuator ::= 'actuator' name '(' params ')' '{' ports '}'

delay ::= 'delay' name  '(' params ')' '{' declarations '}'

function ::= 'function' name  '(' params ')' '{' declarations '}'

capname ::= [A-Z][a-zA-Z0-9_]+

name ::= [a-z][a-zA-Z0-9_]*

qname ::= capname '.' name | name

params ::= (name(','name)+)?

port ::= typename '['expr']' name

ports ::= port (',' port)*

typename ::= name

port_value ::= port '=' expr

port_values ::= port_value (',' port_value)*

declarations ::= declaration*

declaration ::=
    ports_declaration
  | ref_declaration
  | edge_declaration
  | match_declaration
  | edge_precedence_declaration

port_declaration ::= ('in' | 'out') ':' ports ';'

ref_declaration ::= 'let' name '=' qname ('(' expr* ')')? ref_options? ';'

ref_options ::= '[' ref_option* ']'

ref_option ::=
    'abstract'
  | 'period' INT
  | 'parallel' expr
  | 'sequence' expr '(' sequence_ports ')'

sequence_ports ::= name '->' name (',' name '->' name)*

edge_declaration ::= port_path '->' port_paths ';'

port_path ::= name '.' name | name

port_paths ::= port_path (',' port_path)*

match_declaration ::= 'match' name 'with' '|'? match_cases

match_cases ::= match_case ('|' match_case)*

match_case ::= INT '->' declarations

expr ::=  INT | name | expr ('+' | '-' | '*' | '/') expr | '-' expr

edge_precedence_declaration ::= name '-->' name  ';'
```

# Chapter 4

# Defining architectures

Architectures are defined in **.arc** files, using a specific syntax.

Remember that architectures are composed of :

- medium types,
- media, each medium being of a given medium type,
- operator types,
- operators, each operator being of a given operator type,
- gates (of operator types and of operators), each gate having a medium type, determining to which media it can be connected.

An architecture definition is a list of declarations (of medium types, media, operator types or operators).

The names of all these elements are composed of any of the characters **'a'..'z'**, **'A'..'Z'**, **'0'..'9'**, **'_'** and begin with a lowercase letter (**'a'..'z'**).

## 4.1 Medium types, media

**medium types**

A medium type is defined by a kind and a name, with the following syntax :

```
medium type <name> : <kind> [broadcast]
```

**<kind>** can be **SAMPP** (SAM point-to-point), **SAMPP** (SAM multi-point) or **RAM**. The **broadcast** keyword can be added to indicate that the medium works in broadcast mode.

Here are examples of declarations of medium types :

```
medium type tcp : SAMMP
medium type tcpb : SAMMP broadcast
```

**media**

A medium is defined by a name and a medium type. The medium type must be known, i.e. declared before the medium. The syntax is the following :

```
medium <name> : <qualified_medium_type_name>
```

Here are examples of declarations of media :

```
medium mytcp_broadcast : U.tcpb
medium mytcp : U.tcp
```

The example above uses medium types defined in module **U**. See section 4.3 for details about modules.

## 4.2 Operator types, operators

**Operator types**

An operator type is defined by a name and gates, with the following syntax :

```
operator type <name> {
  gate <gate name> : <qualified_medium_type_name> ;
  gate <gate name> : <qualified_medium_type_name> ;
  ...
}
```

The medium type must be known, i.e. declared before the operator type.

Here are examples of declarations of operator types :

```
operator type u {
  gate x : tcp;
  gate y : tcp;
}

operator type ub {
  gate x : tcpb;
}
```

**Operators**

An operator is defined by a name, an operator type and links from gates to media. The operator type and the media linked to the gates must be known, i.e. declared before the operator. The syntax is the following :

```
operator <name> : <qualified_operator_type_name> {
  <gate name> -> <qualified_medium_name> ;
  <gate name> -> <qualified_medium_name> ;
  ...
}
```

The gates are the same as the ones of the operator type. A gate can only be linked to a medium of the same medium type.

Here are examples of declarations of operators :

```
operator p1 : U.u {
  x -> mytcp;
  y -> mytcp;
}

operator p2 : U.ub {
  x -> my_tcp_broadcast;
}
```

The example above uses operator types defined in module **U**. See section 4.3 for details about modules.

## 4.3 Modules

A module system allows separate definition of pieces of architecture. A module contains declarations and can be seen as a library. As soon as a file **foo.arc** contains one declaration (of a medium type, a medium, an operator type or an operator), another architecture file can use the module **Foo** to use one of the elements declared in **foo.arc**.

Refering to a module definition is done simply by prefixing the element name with the module name, as in **U.tcp** which refers to medium **tcp** of module **U**, in the context of a qualified medium name. The same name **U.tcp** could refer to an operator (or another kind of element) in the context of a qualified operator name, if such an operator exists. An example using a module is given in section 2.1.

When a module name appears, the corresponding architecture file is looked up in the *include directories* (specified by **-I** options of the SynDEx tools). The first file named "foo.arc" (in a case-insensitive search) will be considered as the architecture file for module **Foo**. This file is searched in the order in which include directories were specified (except that the file is always searched first in the current directory).

## 4.4 BNF

Here is the architecture specification language represented in Backus-Naur form :

```
arch ::= decl*

decl ::= medium_type | medium | operator_type | operator

medium_type ::= 'medium' 'type' name ':' medium_kind ['broadcast']

medium ::= 'medium' name ':' qname

medium_kind ::= 'SAMPP' | 'SAMMP' | 'RAM'

operator_type ::= 'operator' 'type' name '{' gate_decl+ '}'

operator ::= 'operator' name ':' qname '{' gate_link+ '}'

gate_decl ::= 'gate' name ':' qname ';'

gate_link ::= name '->' qname ';'

capname ::= [A-Z][a-zA-Z0-9_]+

name ::= [a-z][a-zA-Z0-9_]*

qname ::= capname '.' name | name
```

# Chapter 5

# Specifying durations

Adequation heursitics need to know the duration of operations on each operator, and the duration of data communications over each medium. The syntax to specify these durations is described below.

Durations are expressed by integers.

## 5.1 Operation durations

The operation durations, when specified separately (i.e. not in a whole application file, see section **??**), are usually placed in a **.fd** file.

The durations are specified with *rules*, each rule having the form

```
<operation_names> / <operator_type_names> = <integer>
```

with **<operation_names>** being a list of qualified algorithm definition names separated with blanks, and **<operator_type_names>** being a list of qualified operator type names separated with blanks.

Special operation names are allowed to refer to operations created by the flattening with no equivalent in the original algorithm. These operations are :

- **CONDO** to refer to artificially created CondO operations,
- **CONDI** to refer to artificially created CondI operations,
- **EXPLODE** to refer to artificially created Explode operations,
- **IMPLODE** to refer to artificially created Implode operations.

Here is an example of duration specifications :

```
Int.arit_add / U.u U.ub = 2
Int.output Int.input / U.u = 3
Int.arit_mul / U.u = 2
foo CONDO EXPLODE / my_operator_type = 5
```

## 5.2 Communication durations

The communication durations, when specified separately (i.e. not in a whole application file, see section **??**), are usually placed in a **.cd** file.

The durations are specified with *rules*, each rule having the form

```
<type_names> / <medium_type_names> = <integer>
```

with **<type_names>** being a list of type names separated with blanks, and **<medium_type_names>** being a list of qualified medium type names separated with blanks.

Here is an example of duration specifications :

```
bool int uchar / U.tcp U.tcpb = 1
float / U.tcp = 2
int / U.tcp = 2
ushort / U.tcp = 1
```

# Chapter 6

# Specifying placement constraints

Placement constraints consist in authorizing or forbiding the placement of flattened algorithm operations on some operators during the adequation heuristics.

The constraints are usually placed in a **.cts** file.

These constraints are expressed with a list of rules to apply on the current placement constraints, so the order of the rules matters.

Each rule has one of the following forms :

```
<operators> + <operations> ;
<operators> - <operations> ;
```

**operators** is a list of qualified operator names, separated by commas.

**operations** is a list of flattened algorithm operations named with their hierarchy name in the original algorithm. Each name can refer to different elements :

- $d$ where $d$ is a qualified definition name in the original algorithm denotes all operations of the flattened algorithm which correspond to this definition,
- $d.r$ where $r$ is a reference of the definition $d$ in the original algorithm, denotes all operations of the flattened algorithm which correspond to this reference,
- $d.r_1.r_2$, denotes all operations of the flattened algorithm which come from $r_2$, where $r_2$ is a reference of the definition referenced by $r_1$ and $r_1$ is a reference of the definition $d$ in the original algorithm.
- etc.

So, operations in the flattened algorithm are referenced by the name of their original reference or definition in the original algorithm, that is their "hierarchy name" in this original algorithm.

Qualified operator names and operation names can include the special '*' character to match several operator names or operation names. '*' matches 0 to $n$ characters (except '.'). For example,

- **foo*** will match **foo**, **foo1**, **foo_zed** but not **foz** nor **foo.bar**,
- **Int.*** will refer to all operations corresponding to any definition of module **Int**,
- **Int.arit_add.*** will refer to all operations corresponding to any reference of the definition **arit_add** of module **Int**.

The sign between operators and operations specifies the constraint added by the rule :

- '+' will add the ability to distribute the specified operation(s) on the specified operator(s),
- '-' will forbid to distribute the specified operation(s) on the specified operator(s).

At the beginning, all operations can be distributed on all operators. So, with the following example, we forbid to schedule all operations on all operators, then authorize the placement of operation **foo** on **p1** and **bar** and **gee** on **p2** and **p3** :

```
* - * ;
p1 + foo ;
p2, p3 + bar, gee ;
```

# Chapter 7

# Using adequation heuristics

Using adequation heuristics is done with the **syndex** tool (see section 2.2).

Visualization of adequation results can be obtained

- with the **--dot-adeq** option of the **syndex** tool, which dumps a file in graphviz (dot) format,
- or with the **syndex-adeq-gui** tool (see section 2.4) for an interactive graphical display.

# Bibliography

[1] Syndex web site. `http://www.syndex.org`.