Sdrflow User Guide

Dr. János Selmeczi
HA5FT, Jani
ha5ft@freemail.hu

SDRflow

Table of Contents

1General description of *sdrflow*....4
  1.1What is *sdrflow*?....4
  1.2The component architecture....4
  1.3The model of computation....5
  1.4The sdrflow language and the compiler....5
  1.5The development process....6
2The theoretical background of the framework....8
  2.1The data flow paradigm....8
  2.2Actors....8
  2.3Signals....9
  2.4Synchronous data flow graph....10
  2.5Scheduling....11
3The sdrflow language....12
  3.1General properties of the language....12
  3.2Identifiers....12
  3.3Literals....12
  3.4Composite and interface declaration....13
  3.5Declaration of components to be used....14
  3.6Signals....14
  3.7Ports....15
  3.8Signal and port connection rule....17
    3.8.1Class rules....17
    3.8.2Production and consumption rules....18
    3.8.3Type rule....18
    3.8.4Vector size rules....18
    3.8.5Vector count rule....19
  3.9How the signals are seen through ports....19
  3.10Actors....20
  3.11Topology....20
  3.12Schedule....21
4Primitives....22
  4.1Files and directories....22
  4.2Functionalities....23
  4.3Interfacing primitives to the runtime....25
  4.4Pointer management....28

SDRflow

1 General description of *sdrflow*

1.1 What is *sdrflow*?

Sdrflow is a development framework, which could be used to develop signal processing applications for SDR and other purposes. When I started to develop the framework my major goal was to have a system which let me concentrate on writing algorithms and let me forget to deal with the glue code necessary to run the algorithms. It has the following features:

- Component based, and it uses two type of components
  - primitive components written in C language
  - composite components constructed from primitives and other composite components with the sdrflow language.
- The component system is hierarchical with fairly deep hierarchy. The top level component of an application is the application itself.
- The programming model used by the framework is the hierarchical synchronous data flow model.
- Main parts of the framework:
  - the sdrflow language for constructing composite components
  - a compiler and an assembler for compiling composites into binary code
  - a runtime application for running the data flows. The runtime application is lightweight. It could be make run on small, embedded processors, like the ARM Cortex M4.
- The currently published framework runs only on **64 bit Linux systems** (x86 or ARM) , but it implemented in such a way, that in the next release will be able to run on 32 bit Linux systems, and the runtime portion will be adapted for 32 bit ARM processors without any operating system.

The framework is an open source software. It is licensed under GNU GPL 3 or any later versions. It is published on *https://github.com/ha5ft/sdrflow*.

1.2 The component architecture

The framework uses coarse-grained components. Due to this fact the efficiency of the glue code is not critical. This simplifies the code generation and allows the use of a virtual machine to run the glue code.

There are two type of components in the framework.:

- We have primitives which are written in C language and
- we have composites which are constructed from primitives and other composites.

The component system is hierarchical and the framework keeps this hierarchy even at runtime.

I have developed a special language, the sdrflow language for the construction of the composites. In the current release the primitive components

- could be loaded dynamically or
- could be embedded statically

into the runtime system.

The composite code is loaded dynamically in the current release. In the next release the static embedding of the composite code will be possible. This is necessary to run en application on an embedded processor without operating system and file system.

1.3 The model of computation

The framework is based on the synchronous data flow model and not on the dynamic data flow model used in gnu radio or Photos SDR. The synchronous model has some advantages.

- You could precompute the execution schedule. This simplifies the runtime system.
- It enables you the explicit use of feedback loops, which is not possible in the other systems.

I have extended the basic synchronous model. The extensions increase the usability of the model. The most important extensions are:

- the hierarchical description of the system,
- the hierarchical scheduling of the execution,
- the one to many type of connection between the components,
- the explicit use of control parameters in the model,
- separate data flow for computing parameters.

In the next release further extensions will come, like

- a new composite type which conditionally execute it’s components,
- a new composite type which implement a C like switch,
- a new composite type which could iterate over a set of signals.

These extensions are important for creating a full featured SDR application.

1.4 The sdrflow language and the compiler

The sdrflow language is a declarative type language. It is used the describe the composite components. Using the language you only have to declare:

- the components you like to use,
- the instances of the components,
- the signals which connect the components,

- and the topology of the composite, which means how the components are connected by the signals.

To have a feeling here is a sample composite declaration:

```
use  P1
use  C1
composite M1
   context
      input     float[5] i1[]
      output    float[5] o1[]
      parameter int      p1
   end
   signals
      stream    float[5] s1[]
      const     int      c1 273
   end
   actors
      primitive  P1  a1
      composite  C1  a2
      primitive  A7  a7
   end
   topology
      a1.i1  <<  i1
      a1.o1  >>  s1
      a1.p1  <<  p1
      a2.i1  <2< s1
      a2.p1  <<  c1
      a2.o1  >>  o1
   end
   schedule
      auto  a1
   end
end
```

The framework uses a special compiler to translate the composite description into runnable code. The code generated by the compiler runs on a virtual machine.

The language is text based. There is no plan to create a graphical language.

Supported platforms

- **The current release of the framework supports 64 bit Linux systems only**.
- I*n the next release support for 32 bit Linux systems and 32 bit ARM processors without any operating system will come.*

1.5 The development process

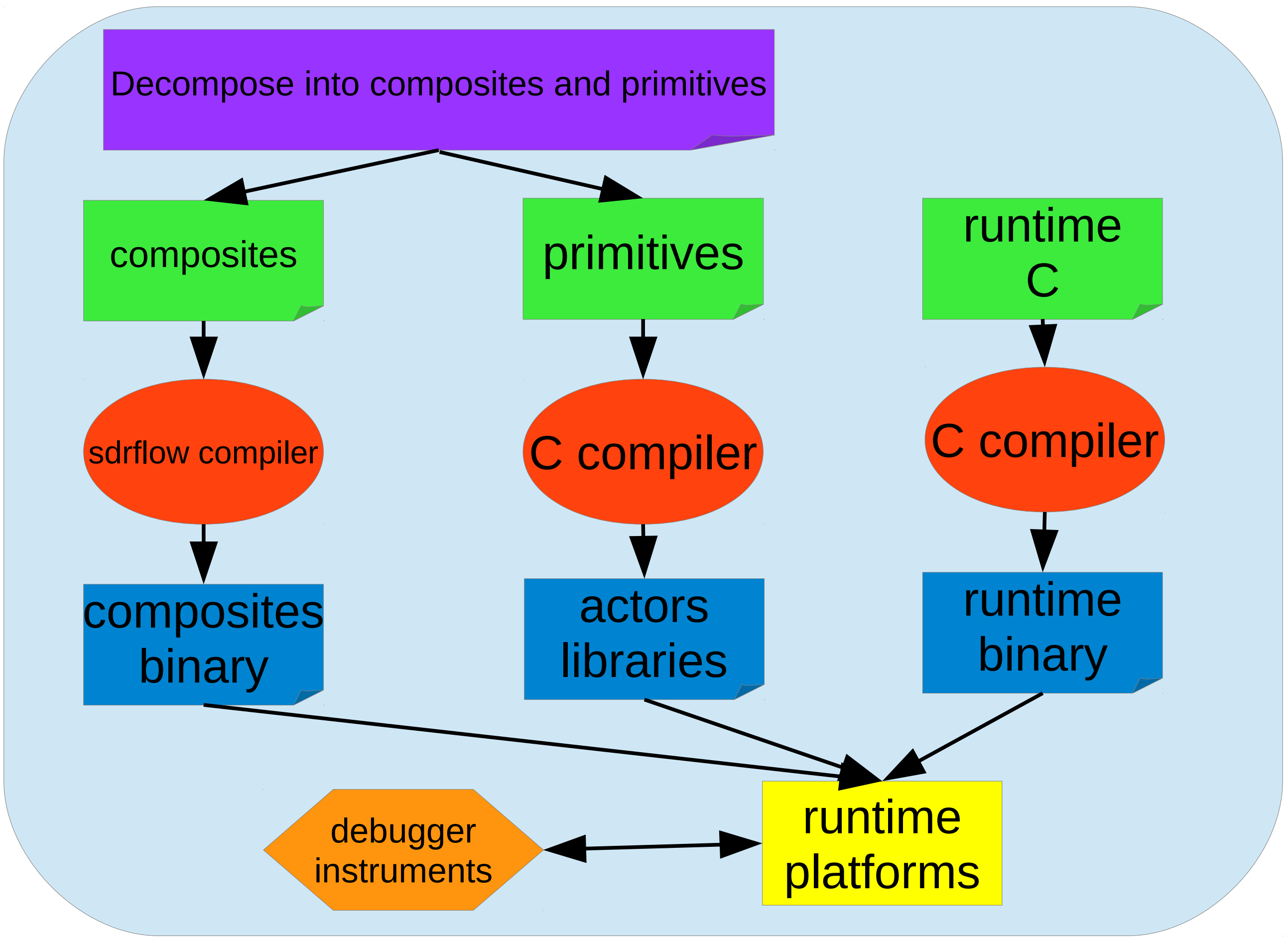
In the development process you are recommended to take the following steps:

- Decompose your signal processing systems into primitive and composite components.

- Write the primitive components in C language and compile them with the gcc compiler into a shared library.
- Write the composite components in the sdrflow language and compile them with the framework’s compiler into a binary code.
- Run the top level composite with the framework’s runtime application.

In the development process you are encouraged to take a step by step approach. You should use test composites to test your primitives and composites.

In the next releases some instrumentation (oscilloscope, signal generator, spectrum analyzer, etc.) will come. With those you could test the system similarly the labor tests of the analog systems.



Pic. 1. This picture shows the development process

2 The theoretical background of the framework

In this part of the user guide I briefly present you the terminology used in connection of data flow systems and some of the theories behind the implementation of my framework.

2.1 The data flow paradigm

In the data flow paradigm we split our program into

- algorithms which do the data processing and
- data management components which manage the data the algorithms are working on.

There are no other code components to deal with. The algorithms will execute whenever they have enough input data. This kind of execution of the algorithms will provide the system functionality.

To define and specify a data flow we should only describe

- the connection between the algorithms and
- the data consuming and producing behavior of the algorithms.

For the description of the system we use a directed graph. The algorithms usually called actors and the data management components called signals.

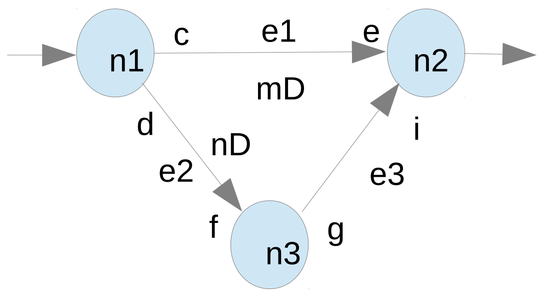


Fig. 2. On this figure a data flow graph is shown. The nodes represent the actors and the edges represent the signals. At the ends of the edges we denotes how many data an actor consumes or produces. for example actor n3 consumes f data elements from signal e2 and produces g data elements to signal e3. The nodes are denoted with the actors names and the edges are denoted with the signals names. On the edges we could note what delays we use in the data transfer. On the figure signal e2 has a delay of n and signal e1 has a delay of m.

2.2 Actors

The instances of the algorithms usually called actors. They are the nodes of the directed graph representing the data flow system. They do atomic execution of their algorithms. This execution is called firing.

They behavior is specified by how many data they consume and produce during a firing. If the production and consumption behavior of an actor is fixed the actor called synchronous.

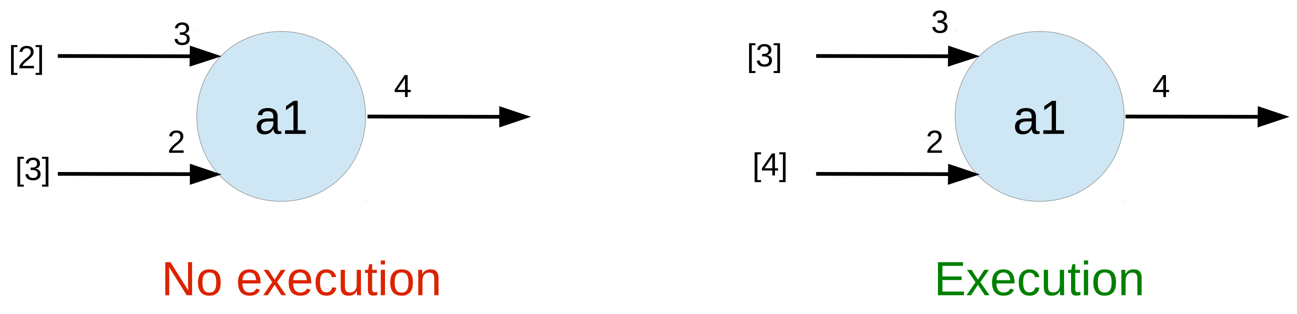


Fig. 3. The figure above shows when an actor is executed and it is not executed. The number at the actor side of the edge shows how many data the actors consume or produce during an execution. The numbers in square brackets shows how many data are stored in the signals. On the left side of the figure the actor a1 could not execute because the upper input signal has only 2 data elements and the actor would need 3 data element for execution. On the right side of the figure the actor has enough data elements at its input for the execution.

2.3 Signals

The instances of the data management components usually called signals. They are the edges of the directed graph. They behave like FIFO buffers with unlimited storage capacity.

In the original model they have a single source and a single destination actor. However I have extended the basic data flow model to allow multiple destination connections.

A signal may have delay. The delay is simple the data elements initially placed in the signal's buffer. Delays some times essential part of the system, because the the signals really delayed. But the delays are useful for solving the problem caused by feedback loops in a data flow system. Using some delays on the feedback signal the system will be schedulable.

I have extended the original signal model of the synchronous data flow graphs. I have introduced three class of signals. The original, FIFO like signal is called stream in my model. There are two more signal classes: the variables a the constants. The variable acts like the variables in the C language and the constants are like constants in C.

If an actor reads the stream in every read it gets a new data elements because the stream represents a series of data elements. In contrast to this if an actor reads a variable or constant it always reads the same data element.

Variables and constants could be used for control parameters in the system.

When computing the execution schedule we do not have to take into account variables and constants. They existence have no effect on the schedule of the system.

The data elements of the signals have types. The framework uses the data types of the C language.

The data elements of the signals are scalars or fixed length vectors of the scalar types.

The stream theoretically consists of an unlimited number of data elements. In practice a stream in the data flow has finite number of data elements. This could be imagined as a sliding window on the series of infinite data elements of the stream. The variables and constants always have finite number of data elements.

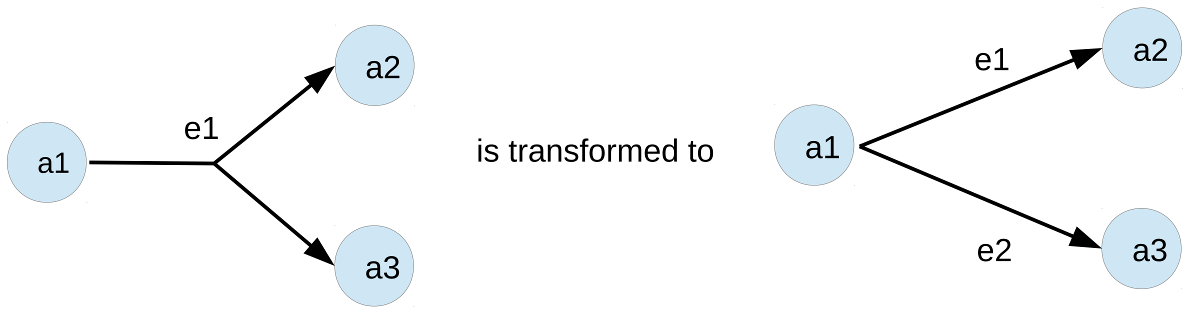


Fig. 4. On this figure you could see how the multi destination edge is transformed into two separate edges in order to have a regular directed graph.

2.4 Synchronous data flow graph

Two actor are said to be connected if a stream class signal connects them. A graph fully connected if we could navigate from any actors to any other actors through stream class signals.

The graph representing the data flow need not be fully connected. They could have several connected subgraphs. To determine the connected subgraphs of the system only the stream class signals have to take into account. So in a connected subgraph you could navigate between any two actors through streams.

If we use variables and constants we could have actors which are connected only by variables and constants or are standalone actors. These actors form a virtual subgraph, which is scheduled differently from the connected subgraphs. It is important, that in this virtual subgraph loops are not allowed. This subgraph usually used to make some computation on the parameters.

If all the actors in the graph are synchronous the graph is called synchronous data flow graph. It has special properties:

- the execution schedule could be precomputed,
- in most of the practical cases they have a periodic schedule, and
- the storage size necessary for stream class signals has an upper bound which could be precomputed too.

The framework’s compiler will use these special properties.

2.5 Scheduling

A schedule is any sequence of the actor’s execution. It is called periodic if for each streams the same number of data elements are produced to and consumed from during the execution of the schedule. This means that in the storage associated with each of the streams the same number of data element are stored before the execution of the schedule and after the execution of the schedule.

The balanced data production and consumption criteria of the periodic schedule gives us a way to compute how many times an actor should be executed in a periodic schedule. We have to solve a set of equations called the balance equations.

A schedule is admissible if whenever an actor is executed in the schedule it has enough input data for the execution. It is very important that if we have admissible schedule we could blindly execute the actors according to the schedule without checking if they have enough data for the execution.

We could find a periodic admissible schedule by solving the balance equations and by simulation after that. There are several algorithm for this.

If a graph with an open feedback loop has an admissible schedule and we put enough delay on the feedback path then the graph with the closed feedback loop will have an admissible schedule.

If we have a schedule we could find repetitive patterns in it and compose nested loops from them. This kind of schedule is called loop schedule.

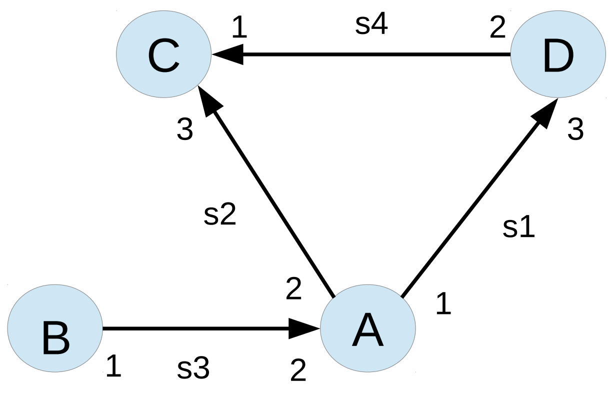


Fig. 5. The figure shows a synchronous data flow graph. The corresponding execution numbers for the actors are: A executed 3 times; B executed 6 times; C executed 1 times; D executed 1 times. The admissible periodic schedule: BBA BBA BBA D C and the same schedule in loop notation: (3((2B)A)DC.

3 The sdrflow language

3.1 General properties of the language

The language is declarative. It does not specify any execution flow. The execution flow is provided by the schedule of the data flow graph.

The language is line oriented. Each line is a sentence. Sentences are built from words. Words are separated with white spaces. Indentation used in the sample code has no syntactic or semantic meaning. It is used for clarity purposes only. If you need longer line (for example for initializing a large vector) you could use line continuation feature similar to the C language or to the shell scripts.

Comments start with semi-colon and end at the end of the line.

3.2 Identifiers

Identifiers are used to name primitives, composites, instances of primitives, instances of composites, signals and ports of primitives and composites.

They are composed from letters, digits, underscores and dots.

The BNF definition of the identifier is the following:

```
letter ::= ("a" | ...... | "z" | "A" |..... | "Z")
digit ::= ("1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
|    "0")
simple_name ::= letter | {letter | digit | "_" | ".")
identifier ::= simple_name | ("_" {"_"} simple_name)
SP ::= (" " | "\t") {" " | "\t")}
NL ::= "\n" | "\r\n"
```

I usually start a primitive or composite name with upper case letter and start the name of the signals, ports and primitive or composite instances with lower case letter.

3.3 Literals

In the sdrflow language we use the C language like literals.

```
Signs ::= ("+" | "=" | "?" | "," | "/" | " " | "_" | "." | ":"
     | "@" | "-")
string ::= """ ((letter | digit | signs) {(letter | digit |
     signs)}) """
unsigned_integer_number ::= digit {digit}
integer_number ::= [("+" | "-")] unsigned_integer
fixed_point_number ::= integer_number "."
unsigned_integer_number
floating_point_number ::= fixed_point_number [("e" | "E")
     integer_number]
literal ::= integer_number | floating_point_number | string
```

3.4 Composite and interface declaration

A composite source code consists of two files

- The file containing the composite declaration. The base name of the file is exactly the same as the identifier of the composite and has an .sdf.src extension. For example for the composite M1 this file is M1.sdf.src. You should write this file for your composite.
- The file containing the interface declaration of the composite. The base name of this file is the identifier of the composite and has and extension of .sdf.ctx. For example for the M1 composite this file is M1.sdf.ctx. For your composite this file is created by the compiler from the composite declaration file.

The primitives has an interface declaration file too with the same naming convention. For the primitives you have to write the interface declaration file.

The interface declaration file contains an interface declaration. The composite declaration file contains the composite declaration.

For declaring an interface you should

- declare if the component is primitive or composite and
- declare all ports of the component using port declaration sentences.

For declaring a composite you should:

- declare all the primitives and composites you use in the new composite using use sentences in the use section,
- declare the composite ports using port declaration sentences in the context section,
- declare the signals using signal declaration sentences in the signal section,
- declare the actor instances using actor declaration sentences in the actor section,
- declare the connection of the actor instances using connection sentences in the topology section,
- declare the scheduling hints using schedule hint sentences in the schedule section.

The BNF description of the composite and interface declaration is the following:

```
composite_declaration ::=
     use_section
     "composite" SP composite_name NL
          context_section
          signal_section
          actor_section
```

```
            topology_section
            schedule_section
        "end" NL

    component_name ::= primitive_name | composite_name
    primitive_name ::= identifier
    composite_name ::= identifier
    interface_declaration ::=
        ("primitive" | "composite") SP component_name NL
            context_section
        "end" NL
```

3.5 Declaration of components to be used

In the composite declaration you have to list all components to be used. The compiler use this list to read the components interface declaration files. You make the listing in the use section of the component.

The syntax of the use section id the following:

```
    use_sentence ::= "use" SP component_name NL
    use_section ::= use_sentence {use_sentence}
```

3.6 Signals

Signals are the data storage elements in a composite.

We have three signal classes:

- the stream class which represent a series of data elements on which the signal processing is working
- the variable class which is similar to the variables in the C language and used for the computation of control parameters
- the constant class which is similar to the constants in the C language and used to fixed value control parameters

The signals consists of fixed length vectors. If the length of the vector is 1 then the vector is a scalar.

The signals have

- Signal class
- Signal type which determine the elementary data element type of the signal from which the vectors of the signal consist of. We use the familiar types of the C language.
- Vector size which specify how many elementary data the vectors of the signal are consist of. It is a fixed number for a signal.

- An identifier (signal name)
- Vector count which determine how many vectors the signal consists of. For signals of the stream class this is really the windows size through which the infinite series of vectors of the stream are seen by the composite.

For a signal declaration you must specify

- the signal class,
- the signal type,
- the vector size if it is larger than 1,
- the signal identifier,
- for variables and constants the vector count if it is larger than 1, for the steams the compiler will do this, so you should use an empty square bracket (“[]”) in place of the vector count.
- for constants and variables the initialization values of the data elements if they are different from 0.

The memory layout of the signals is the same as the memory layout of a 2 dimensional arrays in the C language. The vectors are the rows of the matrix. The vectors follow each others without any gap.

The syntax of the signal generation is the following:

```
signal_class ::= "stream" | "var" | "const"
signal_type ::= "char" | "short" | "int" | "long" | "float" |
     "double" |"uchar" | "ushort" | "uint" | "ulong" |
     "string"
vector_size ::= "["unsigned_integer"]"
vector_count ::= "["unsigned_integer"]" | "[""]"
initializator ::= literal
signal_declaration_sentence ::= signal_class SP
     signal_type[vector_size] SP signal_name
     [vector_count] SP {initializator} NL
signal_section ::= "signals" NL
                         signal_declaration_sentence
                         {signal_declaration_sentence}
                    "end" NL
```

3.7 Ports

Actors connect to signals through ports.

We have three port classes:

- The input class through which the actors get the input data for signal processing. The input ports usually are connected to streams, but this is not necessary. The input porst could be connected to variables and constants too.

- The parameter port through which the actors get the parameters they need to configure they signal processing algorithm. The parameter ports could be connected to variables and constants.
- The output port through which the vectors write the result of they signal processing or write parameters to variables. The output ports could be connected to streams and variables.

The ports have

- Types which determines the scalar type of the data elements accessible through the port. For the port we use the same types we use for the signals. The type of a port and the signal connected to the port should match.
- Vector size which determine the size of the data element vector. If the vector size is 1 then the data element is scalar. The vector size of a port and the signal connected to the port should be the same.
- An identifier (port name)
- Vector count which determine how many data elements (vectors or scalars) the actor during the like to read or write execution of its algorithm through the port. If the port connect to a variable or to a constant signal the vector count of the port and the signal should be the same.
- A delay which specify how many vectors delay the connected signal is accessed with through the port. The delay should be specified in the topology section.

For port declaration you must specify

- the port class,
- the port type,
- the vector size if it is larger then 1,
- the port identifier (port name),
- for all ports of the primitives and for the parameter ports of the composites the vector count if it is larger then 1. The parameter count of the input and output ports of the composite are determined by the compiler, so you should use an empty square bracket pair (“[]”) in place of the vector count.

In a composite declaration the ports of the composite being declared could be used like signals.

- The input and output ports are equivalent to streams. The only difference is that to an input port you could connect only a single component’s input port.
- The parameter ports are equivalent to constants.

Important note: The context section should begin with an input or output port declaration sentence. This make it sure, that the composite will have at least one input or output port.

You should declare ports in the context section of the interface declaration or and in the context section in the composite declaration.

```
port_class ::= "input" | "output" | "parameter"
port_input_or_output_class ::= "input" | "output"
port_type ::= "char" | "short" | "int" | "long" | "float" |
     "double" | "uchar" | "ushort" | "uint" | "ulong" |
     "string"
port_vector_size ::= "["unsigned_integer"]"
port_vector_count ::= "["unsigned_integer"]" | "[""]"
port_name ::= identifier
port_input_or_output_declaration_sentence ::=
     port_input_or_output_class SP
     port_type[port_vector_size] SP port_name
     [port_vector_count] NL
port_declaration_sentence ::= port_class SP
     port_type[port_vector_size] SP port_name
     [port_vector_count] NL
context_section ::=
     "context" NL
          port_input_or_output_declaration_sentence
          {port_declaration_sentence}
     "end" NL
```

3.8 Signal and port connection rule

There are several rules which control how a signal and a port should be connected.

3.8.1 Class rules

Class rules control which port class to which signal class could be connected.

- Input port could be connected to
  - stream signal,
  - variable signal,
  - constant signal,
  - input port of the enclosing composite,
  - parameter port of the enclosing composite.
- Output port could be connected to
  - stream signals,
  - variable signal,
  - output port of the enclosing composite

- Parameter port could be connected
    - variable signal,
    - constant signal,
    - parameter port of the enclosing composite.

3.8.2 Production and consumption rules

These rules control how many producer and consumer could connect to a signal.

- Stream signal should be connected to
    - one and only one output port and
    - at least one input port.
- Variable signal should be connected to
    - one and only one output port and
    - at least one input or parameter port.
- Constant signal should be connected to
    - at least one input or parameter port.
- An input port of the enclosing composite should be connected to
    - one and only one input port
- A parameter port of the enclosed composite should be connected to
    - at least one input or parameter port.
- An output port of the enclosed composite should be connected to
    - one and only one output port.

3.8.3 Type rule

This rule says

- The type of the port and the connected signal should be the same type.
- The type of a port and the connected port of the enclosed composite should be the same type.

3.8.4 Vector size rules

This rule says

- The vector count of the port and the connected signal should be the same number.
- If a port is connected to the port of the enclosing signal then the vector size of the two ports should be the same number.

3.8.5 Vector count rule

This rule controls how the vector count of a port should relate to the vector count of the connected signal.

- If a port of any class is connected to a variable or constant signal the vector count of the port and the vector count of the connected signal should be the same number.
- If a parameter or input port is connected to the parameter port of the enclosing composite then the vector count of the two ports should be the same number.

3.9 How the signals are seen through ports

The actors see the signal through they ports. There are some rules foe this.

- The actors always see vector count number of consecutive vectors through they ports.
- If the port is connected to variable or constant signal or parameter port of the enclosing composite the vector count number of vectors always start from the first vector of the signal. (Important: the first vector is the vector with index 0). This means that in this case the actor always sees all the vectors of the connected signal.
- If the port is connected to a stream or input or output port of the enclosing composite the actor still sees vector count number of vectors, but these vectors will be different vectors in every executions of the actor. It is like the actor sees the stream’s buffer through a moving vector count wide window. In the first execution in a repetition cycle the vectors seen by the actor are the first vector count vectors stored in the stream’s buffer. In each consecutive execution the vector count number wide window will advance by vector count number of vectors.
- If a stream is delayed to an input port:
    - the stream’s buffer is enlarged by delay number of vectors,
    - the actor which reads the signal sees the signal through its input port in the usual ways,
    - the window of the actor which write the signal is offseted in the first execution in a repetition cycle by delay number of vectors from the beginning of the signals buffer. After that the window slides in the usual ways.
    - At the end of the repetition cycle delay number of vectors are copies from the end of the signal’s buffer to the beginning of the buffer.

3.10 Actors

Actors are instances of primitive or composite components. They should be declared in the actor section of a composite declaration.

For an actor declaration you should declare

- if the actor is primitive or composite
- the identifier of the primitive or composite
- the identifier of the actor

The syntax of the actor declaration is the following:

```
primitive_declaration_sentence ::= "primitive" SP
     primitive_name SP actor_instance_name NL
composite_declaration_sentence ::= "composite" SP
     composite_name SP actor_instance_name NL
actor_declaration_sentence ::= (primitive_declaration_sentence
     | composite_declaration_sentence)
actor_instance_name ::= identifier
actor_section ::=
     "actors" NL
          (actor_declaration_sentence)
          {actor_declaration_sentence}
     "end" NL
```

3.11 Topology

The topology means how the actors are connected by signals. It is declared in the topology section.

We use three operators to declare connections. These are:

- the source operator: **`">>"`** , which are used like:

  source **`">>"`** sink

- the sink operator: **`"<<"`** , which are used like:

  sink **`">>"`** source

- the delayed sink operator: **`"<delay<"`** , which are used like:

  sink **`"<delay<"`** source

Rules for using these operators

- On the left side of the operator always is a port.
- On the right side of the operator always is a signal or a port of the enclosing composite.

- On the left side of a delayed sink operator always is an input port.
- On the rigth side of an delayed sink operator always is a stream.
- On the left side of a sink operator alwys is an input or parameter port.
- On the left side of a source operator always id an output port.

The formal syntax is the following:

```
connection_declaration_sentence ::=
     ((actor_port_name SP (">>" | "<<") Sp
          composite_port_name) |
     (actor_port_name SP (">>" | "<<") SP signal_name) |
     (actor_input_port_name SP ("<"delay"<") SP stream_name))
     NL
actor_port_name ::= actor_instance_name"."port_name
actor_input_port_name ::=
actor_instance_name"."input_port_name
input_port_name ::= identifier
stream_name ::= identifier
topology_section ::=
     "topology" NL
          connection_declaration_sentence
          {connection_declaration_sentence}
     "end" NL
```

3.12 Schedule

In the schedule section we declare, which type of schedule we like to have.

We could have two type of schedule

- In case of auto schedule the compiler will compute the schedule for us.
- In case of manual schedule we have to define the schedule, the compiler only check if the schedule is correct.

For both type of schedule we have to specify for which connected subgraph the schedule type should be applied. We specify this with a name of an actor belonging to that subgraph. Most of the cases your composite has only on connected subgraph. In that case you could specify any actors.

Important to note that the execution of the schedules will be in the order the subgraphs are specified. If you have actors which are connected by variables and constants you have a virtual subgraph. You are recommended to specify this subgraph first by any of the actors which are connected by variables or constants only. In this case these actors will be executed first in an repetition cycle. These actors will be executed in the topological order and will be executed only ones in a repetition cycle.

The manual schedule id for those only who like to experiment with scheduling, or for those rare cases when the auto scheduling fails. In mos of the cases you are recommended to use the auto schedule.

The formal description of the schedule section is the following:

```
shedule_element ::= actor_instance_name NL | loop_element
loop_element ::=
     "do" SP unsigned_integer NL
          shedule_element
          {shedule_element}
     "loop"
manual_schedule ::=
     "manual" SP actor_instance_name NL
          schedule_element
          {scedule_element}
     "end" NL
auto_schedule ::= "auto" SP actor_instance_name NL
schedule_section ::=
     "scedule"
          (manual_schedule | auto_schedule)
          {(manual_schedule | auto_schedule)}
     "end"
```

4 Primitives

Primitives are shared libraries which are dynamically loaded into the framework's runtime when a data flow is loaded using the dlload() function.

4.1 Files and directories

Several files and directories are related to a primitive. These have specific names which contains the primitive name the name which will be used for the primitive in the sdrflow language as identifier. In the following we use <primitive_name> for this name.

The important files are (the path names are relative to the framework's sdrflow directory):

actor/<primitive_name>.sdf.so

> This is the shared library file of the primitive. The runtime will look for the shared library files in the actor directory. The build process will put this file into the actor directory.

context/<primitive_name>.sdf.ctx

> This is the interface declaration file of the primitive. The compiler will look for the interface declaration files in the context directory. The build system will put this file into the context directory.

primitive/<primitive_name>/<primitive_name>.c

This file contains the source code of the primitive. The current build system assumes that all the source code is in a single file. If you create your primitive with the build system this file with empty function bodies will be created for you.

primitive/<primitive_name>/<primitive_name>.h

This is the header file for your primitive. It contains structure and type declarations for the primitive. If you create your primitive with the build system this file will be created for you with partly filled bodies of some structures. You should complete the declaration of those structures yourself.

primitive/<primitive_name>/<primitive_name>.sdf.ctx

This is the source file of the interface declaration of the primitive. If you create your primitive with the build system this file will be created for you with an empty context section. You should complete the context section yourself. The build system will copy this file to the context directory.

build/primitive/<primitive_name>/<primitive_name>.o

This is the compiled object code created in the build process.

From the path names you could see which directories are related to a primitive.

4.2 Functionalities

Every primitives should provide five functions. The signature of these function are the same: they get a void pointer and return an integer value. The return value should be 0 if the function has run successfully an nonzero if it failed. The functions are:

int <primitive_name>_load(void *system_catalog) function.

This function is called just after the shared library of the primitive has been loaded. In this function you should do primitive level initialization and primitive level of resource allocation. This initialization a resource allocation are for all instances of the primitive. The load() function gets a pointer to the system_catalog structure.

int <primitive_name>_init(void *context) function.

This function is called every time when the runtime has created an instance of the primitive. In this function you should do instance level initialization and resource allocation. The function gets a pointer to the context structure of the primitive instance which contains pointers to the self structure of the primitive instance and the buffers of the signals connected to the ports of the primitive instance.

int <primitive_name>_fire(void *context) function.

This function is called whenever the instance of the primitive should be executed in the data flow. In this function you should provide the signal processing algorithm of the primitive. The function gets a pointer to the context structure of the primitive instance which contains pointers to the self structure of the primitive instance and the buffers of the signals connected to the ports of the primitive instance.

int <primitive_name>_cleanup(void *context) function.

This function is called just before the runtime will delete an instance of the primitive. It is called before the delete of every instance. In this function you should free all instance level resource which have been allocated in the init() function. The function gets a pointer to the context structure of the primitive instance which contains pointers to the self structure of the primitive instance and the buffers of the signals connected to the ports of the primitive instance.

int <primitive_name>_delete(void *system_catalog) function.

This function is called just before the runtime removes the shared library using the dlclose() function. In tis function you should free all the resources you allocated in the load() function. This function gets a pointer to the system_catalog structure.

If you use the build system to create the files and directories for your new primitive then the empty functions will be created for you in the <primitive_name>.c file. In the following example the primitive name is YourPrimitive.

```
int  YourPrimitive_load(void *sys_catalog)
{
// You should put your own code here
     return 0;
}

int  YourPrimitive_init(void *context)
{
// You should put your own code here
     return 0;
}

int  YourPrimitive_fire(void *context)
{
// You should put your own code here
     return 0;
}

int  YourPrimitive_cleanup(void *context)
{
// You should put your own code here
     return 0;
}
```

```
int  YourPrimitive_delete(void *sys_catalog)
{
// You should put your own code here
     return 0;
}
```

4.3 Interfacing primitives to the runtime

The interfacing of the primitive to the runtime is done with the use of several structures.

<primitive_name>_catalog structure

This is the main interface to the runtime. Each primitive have to provide this structure and the structure should have a specific name <primitive_name>_catalog where <primitive_name> is the same name you use for your primitive in the file names, directory names and in the primitive interface declaration. The runtime will look for this structure using the dlsym() function.

In this structure the primitive provide

- a pointer to the name of the primitive,
- the size of the self structure of the primitive instances and
- the pointers to the five functions of the primitive.

The declaration of the structure is in the include/primitive_interface.h file:

```
typedef int (*primitive_entry_t)(void *context);
struct _primitive_catalog
{
     char                *name;
     size_t              self_size;
     primitive_entry_t   fire;
     primitive_entry_t   init;
     primitive_entry_t   cleanup;
     primitive_entry_t   load;
     primitive_entry_t   delete;
}__attribute__((packed));
typedef struct _primitive_catalog  primitive_catalog_t;
```

The structure is defined in the <primitive_name>.c file. If you use the build system for creating your primitive then the structure definition will be created for you.

```
primitive_catalog_t YourPrimitive_catalog =
{
     .name =        "YourPrimitive",
     .self_size =   sizeof(YourPrimitive_self_t),
     .init =        &YourPrimitive_init,
     .fire =        &YourPrimitive_fire,
     .cleanup =     &YourPrimitive_cleanup,
     .load =        &YourPrimitive_load,
     .delete =      &YourPrimitive_delete
};
```

system_catalog structure

The system catalog is a facility with which the runtime could provide callback functions for the primitives. This is useful, if like to run the data flow on a processor without operating system. In that case the runtime could provide special functionality beyond the standard C language libraries. In the current release we only demonstrate this possibility, but we pass NULL pointers as function pointers.

The structure declaration in the include/primitive_interface.h file: Important: the callback functions do not exist, NULL pointers are passed in the structure in the current release.

```
typedef int (*open_cmd_channel_t)(char *name);
typedef int (*read_cmd_channel_t)(int channel, char *cmd,
          int size);
typedef int (*close_cmd_channel_t)(int channel);

struct _system_catalog
{
//   char *program_name;
     int       version;
     open_cmd_channel_t  open_cmd;
     read_cmd_channel_t  read_cmd;
     close_cmd_channel_t close_cmd;
}__attribute__((packed));

typedef struct _system_catalog    system_catalog_t;
```

<primitive_name>_context structure

Using this structure, the runtime passes

- pointer to the self structure of the primitive
- pointers to the buffers of the signals connected to the ports of the primitive.

Pointer to this structure is passed to the init(), fire() and cleanup() functions of the primitives.

The declaration of the structure should match the declaration of the primitive interface.

If the declaration of the primitive interface in the YourPrimitive.sdf.ctx file id the following:

```
primitive YourPrimitive
     context
          input     float[2048]    inp[1]
          output    double[2048]   out[1]
          parameter int  param1
     end
end
```

then the declaration of the context structure in the YourPrimitive.h file should be the following:

```
struct _YourPrimitive_context
{
     YourPrimitive_self_t     *const self;
     float                    *const inp;
     double                   *const out;
     int                      *const param1;
}__attribute__((packed));
typedef struct _YourPrimitive_context
     YourPrimitive_context_t;
```

You are recommended to declare the pointers as constant pointers. It is not necessary, but it is for your safety.

<primitive_name>_self structure

This structure could be used to save data from one execution of the fire() function to the next executions. You could save the instance level resources in this structure too.

The runtime provide you the instance name of your primitive in this structure. In the current relese the instance name is the identifier of the instance as declared in the sdrflow language code. In the next release it will be a path name which will show where the hierarchy the instance is. This pathname will be unique in the runtime.

You should declare this structure in the <primitive_name>.h header file. If you create your primitive with the build system, a partially declared structure containing only the instance name pointer will be provided for you.

An example from the SignalGen.h file

```
struct    _signalgen_self
{
     char      *instance_name;
     float     freq;
     float     gain;
     double    phase;
```

```
    double    phase_increment;
};

typedef   struct _signalgen_self   signalgen_self_t;
```

4.4 Pointer management

Primitives access signal data through the pointers in the context structure. These pointers are declared for the primitive as constant pointers. The primitive are not allowed to change them. The primitive could use them with array indexing or assign them to a not constant pointer. The runtime manages the values of this pointers. The management is done according to the rules in chapter 3.9.

- The primitive always can access

  vector_size * vectors_count * sizeof(port_type)

  bytes memory beginning with the address the pointer points to. Here vector_count, vector_size and port_type is related to the port the pointer corresponds to.

- If the pointer corresponds to a port which is connected to variable or constant signal or parameter port of the enclosing composite the accessible memory always starts at the beginning of the buffer of the signal. At the beginning of a repetition cycle the runtime always resets the pointer value to the address of the beginning of the buffer.

- If the pointer corresponds to a port which is connected to a stream or input or output port of the enclosing composite the accessible memory slides through the buffer of the signal. At the beginning of the repetition cycle the pointer is set to point to the beginning of the buffer of the signal. After each execution of the primitive fire() function the runtime increases the pointer value with

  vector_size * vectors_count * sizeof(port_type)

  bytes.

- If a stream is delayed to an input port:
  - the stream's buffer is enlarged by (delay * vector_size * sizeof(signal_type)) bytes.
  - the pointer which corresponds to the input port is managed in the usual ways,
  - the pointer which corresponds to a port which is connected to a delayed signal at the beginning of a repetition cycle is set to (addressof(beginning_of_the_buffer) + (delay * vector_size * sizeof(signal_type))). After each execution of the primitive fire() function the pointer is increased in the usual ways..
  - At the end of the repetition cycle (delay * vector_size * sizeof(signal_type)) number of bytes are copied from the end of the buffer to the beginning of the buffer.