C/C++ Coding Guide

Kai Huang and Wolfgang Haid
Revision 1313, May 6, 2009

This document describes how to program DOL applications using C/C++ as programming language.

To be able to achieve a good performance of applications running on the SHAPES architecture platform, a certain coding style is required to program these applications. This coding style has been devised for the SHAPES project in which application code is expected to be programmed once but can be used without any modification for different tool chains, e.g. functional simulation, hardware dependent software generation, fast simulation, and final system synthesis. Another key aspect of the SHAPES project is the design space exploration (DSE) used for generating a mapping of the application onto the SHAPES architecture platform. To enable semi-automatic or automatic DSE, an efficient extraction of key parameters of the application is required. Both aspects necessitate consistent code and a well-defined application programming interface.

In the SHAPES project, an application is described as a process network where processes that are connected by channels perform the application's function. This document describes how to implement these processes and the communication between processes. The key properties of the coding style are:

- The internal state of a process is stored in a structure. The details about this structure are given in section 1.
- The behavior of a process is abstracted into two methods, namely `DOL_init()` and `DOL_fire()`. The function `DOL_detach()` can be used to terminate the execution of a process. The details about these functions are given in section 2.
- The communication between processes is unified into a set of four functions, i.e. `DOL_read()`, `DOL_write()`, `DOL_rtest()`, `DOL_wtest()`. The details about these functions are given in section 3.

For illustration and reference, an example is shown in section 5.

1 Process Structure

The interface to the internal state and the behavioral functions of a process is abstracted in the structure `DOLProcess`. This structure `DOLProcess` can be seen as an interface in the object-oriented sense that allows a unified access to any process, independent of the actual process-specific implementation.

`DOLProcess` contains four pointers, namely `local`, `init`, `fire`, and `wptr`, as shown in the listing 1:

- The `local` member, line 13, is a pointer to a structure that will store local information of a certain process. "Local" means that the data of the referenced structure is visible for just one process instance. If a process is iterated, each instance will have its own local structure.
- The `init` and `fire` function pointers, lines 14 and 15, point to the corresponding behavior functions which are described in section 2. that each process can have its own behavior.
- The `wptr` pointer, line 16, is a place holder for tool-chain specific process information. In the DOL SystemC functional simulation, for instance, `wptr` is used to point to an instance of the C++ process class wrapper.

```
01 #ifndef DOL_H
02 #define DOL_H
03
04 struct _process;
05 typedef struct _local_states *LocalState;
06 typedef void *WPTR;
07
08 typedef void (*ProcessInit)(struct _process*);
09 typedef int (*ProcessFire)(struct _process*);
10
11 // process handler
12 typedef struct _process {
13     LocalState      local;
14     ProcessInit     init;
15     ProcessFire     fire;
16     WPTR            wptr;
17 } DOLProcess;
18
19 #endif
```

Listing 1: Header File dol.h

2 Behavior Function

The functional behavior of a process is split up into two phases, an initialization phase and an execution phase, which are defined in the two corre-

sponding functions `init()` and `fire()`. Users are, however, free to add other methods, which can be called from `fire()` and `init()`. The function `DOL_detach()` can be used to terminate a process.

- The `DOL_init()` function of each process is called once at the initialization of the process network. The purpose of this function is to initialize a process before running.
- The `DOL_fire()` function will be called repeatedly by the scheduler during runtime, by which the actual behavior of a process is activated.
- `DOL_detach()` signals the scheduler that this process must not be executed again.

`DOL_init()` and `DOL_fire()` must not contain infinite loops. For interprocess communication, the communication functions described in the next section can be used inside the `DOL_init()` and `DOL_fire()` function.

3 Communication Interface

All processes use uniform communication primitives, i.e. `DOL_read()`, `DOL_write()`, `DOL_rtest()`, and `DOL_wtest()`.

- `int DOL_write(void *port, void *buf, int len, DOLProcess *p)` attempts to write up to `len` bytes from port descriptor `port` into the buffer starting at `buf`.
- `int DOL_read(void *port, void *buf, int len, DOLProcess *p)` attempts to read up to `len` bytes from port descriptor `port` into the buffer starting at `buf`.
- `int DOL_rtest(void *port, int len, DOLProcess *p)` checks whether the space of connected FIFO is larger than `len`. If yes, return `1`, otherwise return `0`;
- `int DOL_wtest(void *port, int len, DOLProcess *p)` checks whether the data in the connected FIFO is larger than `len`. If yes, return `1`, otherwise return `0`;

The syntax of `DOL_read()` and `DOL_write()` primitives is the same as that of the POSIX `read()`/`write()`, except one additional parameter, i.e. `DOLProcess *p`, which stores the address of the caller process. The semantics of these primitives depend on the definition of the channel connected and could be implemented in different manners with respect to different code

generators. For instance, when all channels are FIFOs, then the SystemC functional simulation generator creates pure FIFO semantics C++ code. In addition to the blocking `DOL_read()` and `DOL_write()`, there are two functions that can be used to implement non-blocking reads and writes, namely `DOL_rtest()` and `DOL_wtest()`. These two functions are non-blocking test primitives that can be used to test the status of a FIFO channel, e.g. full or empty. For an output port which connects to the input of a FIFO, only DOL_wtest() is defined. The same definition applies to the input port of a process.

3.1 Port Access

The first argument of all communication primitives is `void *port`. The following rules apply for this parameter. There is a difference between accessing an ordinary port, that is, a port that is a direct child element of its enclosing process element, and an iterated port, that is, a port that is created using an iterator element within the process element.

Ordinary (Not Iterated) Ports

- For each port which is accessed within a process, a port macro must be declared in the header file of the process. This declaration must take the following form:

  ```
  01 #define PORT_X portname
  ```

  The name of the port must be the same as defined in the process network XML file. If the name of the port is an integer (as required by HdS generation), `portname` is simply the corresponding integer. If the name of the port is a string, `portname` is the string in quotes. The name of the defined macro must have the prefix "`PORT_`". It is prohibited to define any other macro having "`PORT_`", "`INPORT_`", or "`OUTPORT_`" as its prefix.

- Within a call to a communication primitive, only the defined macro may be used, not the string itself:

  ```
  01 DOL_read((void*)PORT_X, &buffer, 4, p);
  ```

Iterated Ports

- For each iterated port which is accessed within a process, a port macro must be declared in the header file of a process. This declaration takes

the same form as for an ordinary port, but the name of the port is now the basename of the iterated port as specified in the process network XML file:

```
01 #define PORT_Y portbasename
```

- In the C file, use the `CREATEPORTVAR(buffer)` macro (which has a tool-chain specific implementation) to create a buffer where the reference to an iterated port can be saved. The variable name can be chosen arbitrarily (as long as it does not conflict with any other variable):

```
01 CREATEPORTVAR(port);
```

- In the C file, use the `CREATEPORT()` macro (which also has a tool-chain specific implementation) to create a reference to an iterated port and save it in the previously generated buffer. The signature of the `CREATEPORT()` macro is as follows:
`CREATEPORT(buffer, basename, numberOfIteratorDimensions, index0, range0, [index1, range1, [index2, range2, [index3, range3]]])`

  - `buffer` is the buffer created by `CREATEPORTVAR(buffer)`.
  - `basename` is the basename of the port. Similar to the rule for the ordinary port, only the defined macro may be used.
  - `numberOfIteratorDimensions` is the number of dimensions of this port, that is, the number of nested iterator parent elements.
  - `index0` gives the index of the port in the first dimension. The same applies for `index1`, `index2`, `index3` for the other (up to four) dimensions.
  - `range0` is the range of the iterator in the first dimension. The same applies for `range1`, `range2`, `range3` for the other (up to four) dimensions.

  An example is shown below:

```
01 CREATEPORT(port, PORT_Y, 2,
02              row, NUMBER_OF_ROWS, col, NUMBER_OF_COLS);
```

- Use the variable specified in `CREATEPORT()` within a call to a communication primitive:

```
01 DOL_read((void*)port, &buffer, 4, p);
```

As an example for demonstrating the usage of the mentioned macros, the *output_consumer* process from Example 4 is taken.

```
<process name="output_consumer">
  <iterator variable="row" range="for (int row = 0; row &lt; N; row++)">
    <iterator variable="col" range="for (int col = 0; col &lt; N; col++)">
      <port type="input" name="matrixC">
        <append function="row"/>
        <append function="col"/>
      </port>
    </iterator>
  </iterator>
  <source type="c" location="consumer.c"/>
</process>
```

Listing 2: Example for Process With Iterated Ports

```
#define PORT_MATRIXC "matrixC"

int consumer_fire(DOLProcess *p) {
  double matrixC_value;
  int row, col;
  CREATEPORTVAR(port);

  for (row = 0; row < NUMBER_OF_ROWS_COLS; row++) {
    for (col = 0; col < NUMBER_OF_ROWS_COLS; col++) {
      CREATEPORT(port, PORT_MATRIXC, 2,
                 row, NUMBER_OF_ROWS_COLS,
                 col, NUMBER_OF_ROWS_COLS);
      DOL_read((void*)port, &matrixC_value, sizeof(double), p);
    }
  }
  return 0;
}
```

Listing 3: Example for Reading From Iterated Ports

4 Further Rules

Besides the guidelines described above, there are several additional rules. and platform-specific rules.

4.1 General Rules

These following rules apply to program code that should be executed either using functional simulation or on the real or virtual platform.

- The structure for saving the user-defined local data of a process needs to be named complying with the following naming convention: `Processname_State`, that is, the name of the process with the first letter capitalized followed by `_State`.

- When a function has a pointer to a `DOLProcess` structure as an argument, such as `square_init()` and `square_fire()` in the example in section 5, the argument name needs to be "`p`".

- Iterated processes have a similar behavior and, therefore, possess the same source code file. Nevertheless, sometimes it is desirable that these processes behave slightly different depending on their position in the process network. The macro `GETINDEX(dim)` can be used to retrieve the iteration index of a process in the `dim`-*th* dimension.

4.2 Platform-Specific Rules

The following rules are restrictions arising from the particularities of the SHAPES hardware platform and its software tool-flow.

- Use `long` as datatype for integer variables and `float` as datatype for floating-point variables.

- Due to different word length of datatypes on the different processors of the SHAPES platform, it is not possible to write or read variables (or structures) with different datatypes on the same software channel. This means that each channel can only be used to convey single values or arrays of values of one datatype between processes. If processes need to exchange values of multiple kinds of datatypes, multiple channels need to be used.

- In order to enable automated profiling of an application on the virtual SHAPES platform, an application needs to terminate after a certain amount of time. Specifically, this means that each process needs to call `DOL_detach()` to indicate its termination. If `DOL_detach()` is not called by *all* processes, automated profiling will not work.

5 Example

An example implementation of a process complying with the described coding style is shown in the listings 4 and 5. In particular, these listings show the header and the C file of the *square* process of Example 2.

In the header file, the ports of this process, the local data structure, and the behavior functions are declared, as shown in lines 06–14. In the corresponding C file, the behavior functions are defined.

```
01 #ifndef SQUARE_H
02 #define SQUARE_H
03
04 #include <dol.h>
05
06 #define PORT_INPUT  0
07 #define PORT_OUTPUT 1
08
09 typedef struct {
10     int index;
11 } Square_State;
12
13 void square_init(DOLProcess *);
14 int square_fire(DOLProcess *);
15
16 #endif
```

Listing 4: Header File of *square* Process

```
01 #include <stdio.h>
02 #include "square.h"
03
04 void square_init(DOLProcess *p) {
05     ((Square_State*)p->local)->index = 0;
06 }
07
08 int square_fire(DOLProcess *p) {
09     float i;
10
11     if (((Square_State*)p->local)->index < LENGTH) {
12         DOL_read(PORT_INPUT, &i, sizeof(float), p);
13         i = i*i;
14         DOL_write(PORT_OUTPUT, &i, sizeof(float), p);
15         ((Square_State*)p->local)->index++;
16     }
17
18     if (((Square_State*)p->local)->index >= LENGTH) {
19         DOL_detach(p);
20     }
21
22     return 0;
23 }
```

Listing 5: C File of *square* Process