

一. 实验题目

DOL实例分析&编程

二. 实验目的

1. 理解DOL运行方式
2. 进行DOL编程

三. 实验内容

DOL代码位置和说明

- dol/examples/exampleX
 - /src文件夹内包含2种文件：.c, 与对应的.h, 就是实现的模块，就是.dot的框框的功能描述。（每个模块要实现2个接口，xxx_init和xxx_fire两个函数，分别是初始化这个模块是干了什么，以及这个模块开干的时候做什么）。定义进程：每个模块都要写上xxx_fire（可能被执行无数次），至于init是可选择写或者不写的，xxx_init（只会被执行一次）
 - example*.xml 里面定义了模块与模块之间是怎么连接的，就是有哪些框，哪些线，比如A框跟B框用一根线连起来，他们就在一起了。这个xml是这样的：process就是那些框，sw_channel那些线，connection就是把线的那头连到框的那头。

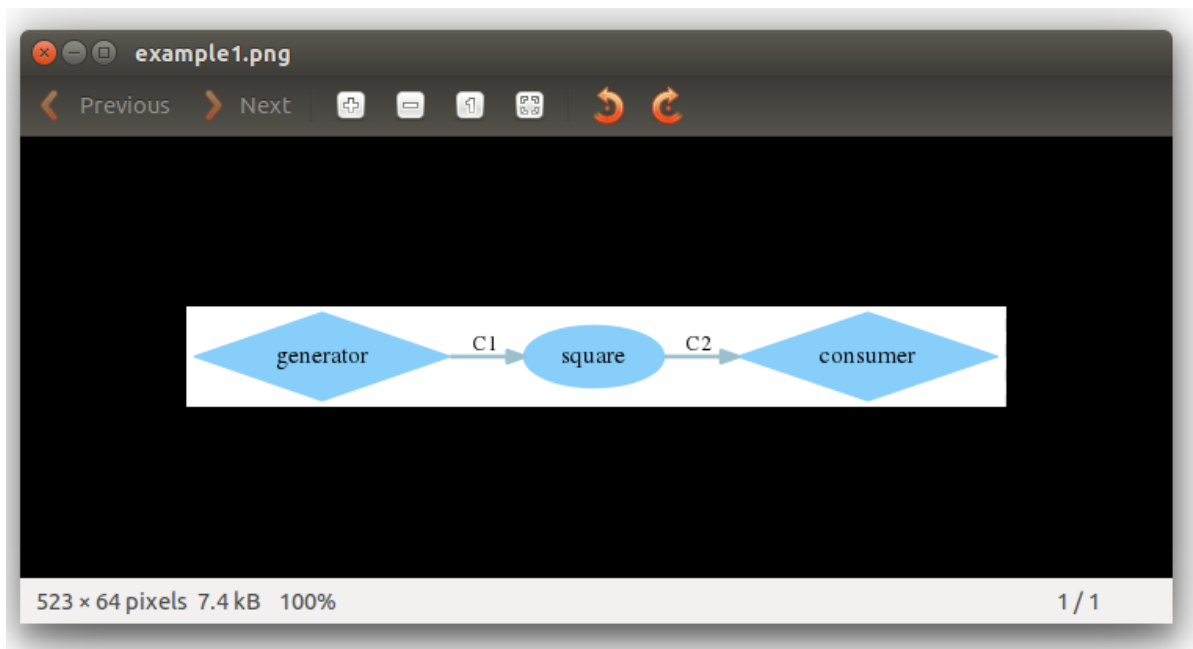
实验任务

1. 修改example2，让3个square模块变成2个，tips:修改xml的iterator
2. 修改example1，使其输出3次方数，tips:修改square.c

四. 实验过程

1. example1 代码分析和修改

- 首先我们看到example1之后的dot图，其中包含生产者、平方模块、消费者（3个模块）通道C1与C2（两条线）



- 首先是generator.h, 这个是定义了一个生产者结构体和声明了一下函数就不看了
- 接着是generator.c

```
#include <stdio.h>
#include <string.h>
#include "generator.h"
// initialization function
void generator_init(DOLProcess *p) {
    p->local->index = 0;
    p->local->len = LENGTH;
}
```

generator_init是初始化函数。这里代码的意思是将当前位置置为0, 设置生产者长度。这里的local指针指向的是.h文件的_local_states结构。

```
int generator_fire(DOLProcess *p) {
    if (p->local->index < p->local->len) {
        float x = (float)p->local->index;
        DOL_write((void*)PORT_OUT, &(x), sizeof(float), p);
        p->local->index++;
    }
    if (p->local->index >= p->local->len) {
        DOL_detach(p);
        return -1;
    }
    return 0;
}
```

generator_fire是信号产生函数。如果当前位置小于生产长度, 则将x (这里是当前下标) 写入到输出端, 大于等于x时销毁进程生产者进程。所以这个函数就只是执行length次生产送数, 之后自动销毁。

- 然后consumer.h 也是定义结构体和声明了一下函数

- o consumer.c

```
#include <stdio.h>
#include "consumer.h"
void consumer_init(DOLProcess *p) {
    sprintf(p->local->name, "consumer");
    p->local->index = 0;
    p->local->len = LENGTH;
}
```

consumer_init初始化函数，含义同generator_init

```
int consumer_fire(DOLProcess *p) {
    float c;
    if (p->local->index < p->local->len) {
        DOL_read((void*)PORT_IN, &c, sizeof(float), p);
        printf("%s: %f\n", p->local->name, c);
        p->local->index++;
    }
    if (p->local->index >= p->local->len) {
        DOL_detach(p);
        return -1;
    }
    return 0;
}
```

consumer_fire信号消费函数，若当前位置小于设定长度，则读出输入端信号，并且打印；到达设定长度后销毁消费者进程。

- o square.h 还是定义结构体和声明了一下函数
- o square.c平方函数

```
#include <stdio.h>
#include "square.h"
void square_init(DOLProcess *p) {
    p->local->index = 0;
    p->local->len = LENGTH;
}
int square_fire(DOLProcess *p) {
    float i;
    if (p->local->index < p->local->len) {
        DOL_read((void*)PORT_IN, &i, sizeof(float), p);
        i = i*i;
        DOL_write((void*)PORT_OUT, &i, sizeof(float), p);
        p->local->index++;
    }
    if (p->local->index >= p->local->len) {
        DOL_detach(p);
        return -1;
    }
    return 0;
}
```

```
}
```

square_init同上面一堆init一样，square_fire信号处理函数，读入输入端信号i，将其平方后写出到输出端，也是重复length次之后就停止。

- o 然后是example1.xml

```
<!-- processes -->
<process name="generator">
  <port type="output" name="1"/>
  <source type="c" location="generator.c"/>
</process>

<process name="consumer">
  <port type="input" name="1"/>
  <source type="c" location="consumer.c"/>
</process>

<process name="cube">
  <port type="input" name="1"/>
  <port type="output" name="2"/>
  <source type="c" location="cube.c"/>
</process>
```

process name: 定义进程名字，也就是对应的src中的.c和.h的名字

port: type定义端口 输入or输出 name 定义端口名字

source : type 定义语言类型 location定义文件位置

```
<!-- sw_channels -->
<sw_channel type="fifo" size="10" name="C1">
  <port type="input" name="0"/>
  <port type="output" name="1"/>
</sw_channel>

<sw_channel type="fifo" size="10" name="C2">
  <port type="input" name="0"/>
  <port type="output" name="1"/>
</sw_channel>
```

通道定义，一条线就是一条通道

sw_channel 其中的type定义类型，size定义缓冲区大小 name定义线的名字

```
<!-- connections -->
<connection name="g-c">
  <origin name="generator">
    <port name="1"/>
  </origin>
  <target name="C1">
    <port name="0"/>
  </target>
</connection>
```

```

    </target>
</connection>

<connection name="c-c">
    <origin name="C2">
        <port name="1"/>
    </origin>
    <target name="consumer">
        <port name="1"/>
    </target>
</connection>

<connection name="s-c">
    <origin name="cube">
        <port name="2"/>
    </origin>
    <target name="C2">
        <port name="0"/>
    </target>
</connection>

<connection name="c-s">
    <origin name="C1">
        <port name="1"/>
    </origin>
    <target name="cube">
        <port name="1"/>
    </target>
</connection>

```

connections定义各个模块之间的连接,一条线会对应两个connection。主要就是 从origin 的某个port 连接到 target 某个port。

- 此处实验任务2修改example1, 使其输出3次方数, tips:修改square.c

就是要修改square_fire中的 $i=i*i$ 即可, square_fire中的其他部只是输入输出和判断截止所以我们修改后的square_fire函数为

```

int square_fire(DOLProcess *p) {
    float i;
    if (p->local->index < p->local->len) {
        DOL_read((void*)PORT_IN, &i, sizeof(float), p);
        i = i*i*i;
        DOL_write((void*)PORT_OUT, &i, sizeof(float), p);
        p->local->index++;
    }
    if (p->local->index >= p->local->len) {
        DOL_detach(p);
        return -1;
    }
    return 0;
}

```

这样子就可以运行出如下结果

```
execute:
[echo] Make HdS application.
[exec] make: Nothing to be done for `all'.
[echo] Run HdS application.
[concat] consumer: 0.000000
[concat] consumer: 1.000000
[concat] consumer: 8.000000
[concat] consumer: 27.000000
[concat] consumer: 64.000000
[concat] consumer: 125.000000
[concat] consumer: 216.000000
[concat] consumer: 343.000000
[concat] consumer: 512.000000
[concat] consumer: 729.000000
[concat] consumer: 1000.000000
[concat] consumer: 1331.000000
[concat] consumer: 1728.000000
[concat] consumer: 2197.000000
[concat] consumer: 2744.000000
[concat] consumer: 3375.000000
[concat] consumer: 4096.000000
[concat] consumer: 4913.000000
[concat] consumer: 5832.000000
[concat] consumer: 6859.000000

BUILD SUCCESSFUL
Total time: 3 seconds
root@15352008蔡荣裕:~/dol/build/bin/main#
```

确认一下这个的确是立方。但是这个结果输出的还是之前的那个图，觉得吧不太好，所以魔改一波。

- 我们把square重写一下重写为cube。

cube.h如下：

```
#ifndef CUBE_H
#define CUBE_H

#include <dol.h>
#include "global.h"

#define PORT_IN 1
#define PORT_OUT 2

typedef struct _local_states {
    int index;
    int len;
} Cube_State;

void cube_init(DOLProcess *);
int cube_fire(DOLProcess *);

#endif
```

这里主要就是注意一下改名尤其是那个Cube_State，之后运行命令会去认他，所以这个改名最为重要，函数倒是可以不改的，但是改了好点。

cube.c如下

```
#include <stdio.h>

#include "cube.h"

void cube_init(DOLProcess *p) {
    p->local->index = 0;
    p->local->len = LENGTH;
}

int cube_fire(DOLProcess *p) {
    float i;

    if (p->local->index < p->local->len) {
        DOL_read((void*)PORT_IN, &i, sizeof(float), p);
        i = i*i*i;
        DOL_write((void*)PORT_OUT, &i, sizeof(float), p);
        p->local->index++;
    }

    if (p->local->index >= p->local->len) {
        DOL_detach(p);
        return -1;
    }

    return 0;
}
```

- 然后我们修改一下xml文件因为我们改了模块名字那些连线框什么的也一起改修改，只需要把原来square改写为cube，也就processes和connections两部分

```
<!-- processes -->
<process name="cube">
    <port type="input" name="1"/>
    <port type="output" name="2"/>
    <source type="c" location="cube.c"/>
</process>

<!-- connections -->

<connection name="s-c">
    <origin name="cube">
        <port name="2"/>
    </origin>
    <target name="C2">
        <port name="0"/>
    </target>
</connection>
```

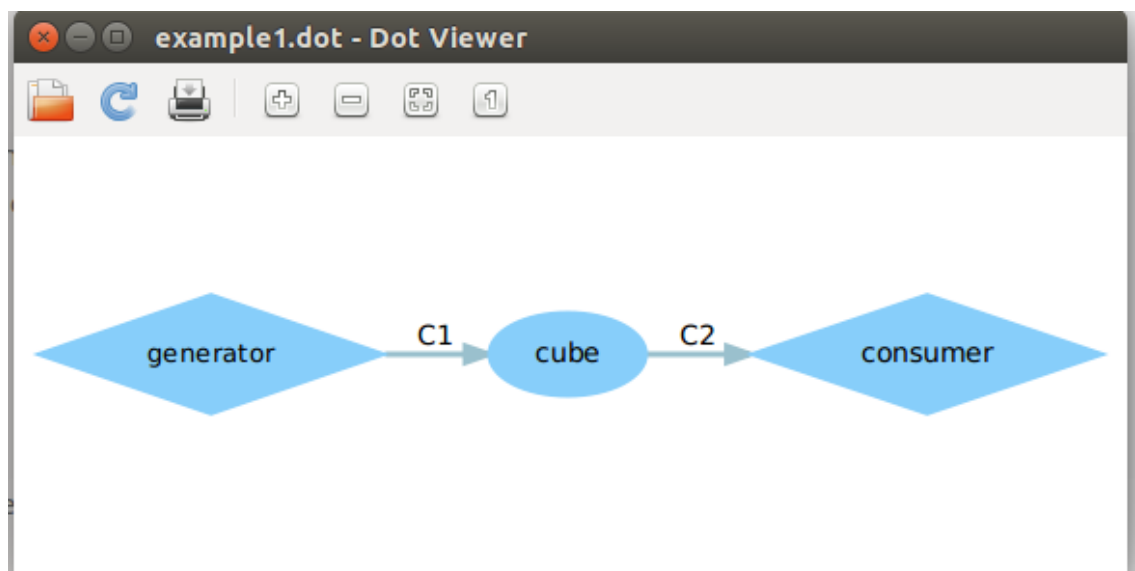
```

<connection name="c-s">
  <origin name="C1">
    <port name="1"/>
  </origin>
  <target name="cube">
    <port name="1"/>
  </target>
</connection>

</processnetwork>

```

- 重新build一下，运行结果如下：



这个算是彻底改完了。

2. example2 代码分析和修改

- 代码好像没什么了全部和example1一样。
- 直接看到xml文件

```

<variable value="3" name="N"/>

<!-- instantiate resources -->
<process name="generator">
  <port type="output" name="10"/>
  <source type="c" location="generator.c"/>
</process>

<iterator variable="i" range="N">
  <process name="square">
    <append function="i"/>
    <port type="input" name="0"/>
    <port type="output" name="1"/>
    <source type="c" location="square.c"/>
  </process>
</iterator>

```



```

<process name="consumer">
  <port type="input" name="100"/>
  <source type="c" location="consumer.c"/>
</process>

```

可以看到此处通过迭代定义了3个square的模块其他的都一样和example1一致。

```

<iterator variable="i" range="N + 1">
  <sw_channel type="fifo" size="10" name="C2">
    <append function="i"/>
    <port type="input" name="0"/>
    <port type="output" name="1"/>
  </sw_channel>
</iterator>

```

此处也是通过迭代建立了4条连接

```

<iterator variable="i" range="N">
  <connection name="to_square">
    <append function="i"/>
    <origin name="C2">
      <append function="i"/>
      <port name="1"/>
    </origin>
    <target name="square">
      <append function="i"/>
      <port name="0"/>
    </target>
  </connection>

  <connection name="from_square">
    <append function="i"/>
    <origin name="square">
      <append function="i"/>
      <port name="1"/>
    </origin>
    <target name="C2">
      <append function="i + 1"/>
      <port name="0"/>
    </target>
  </connection>
</iterator>

```

通过迭代进行连线。

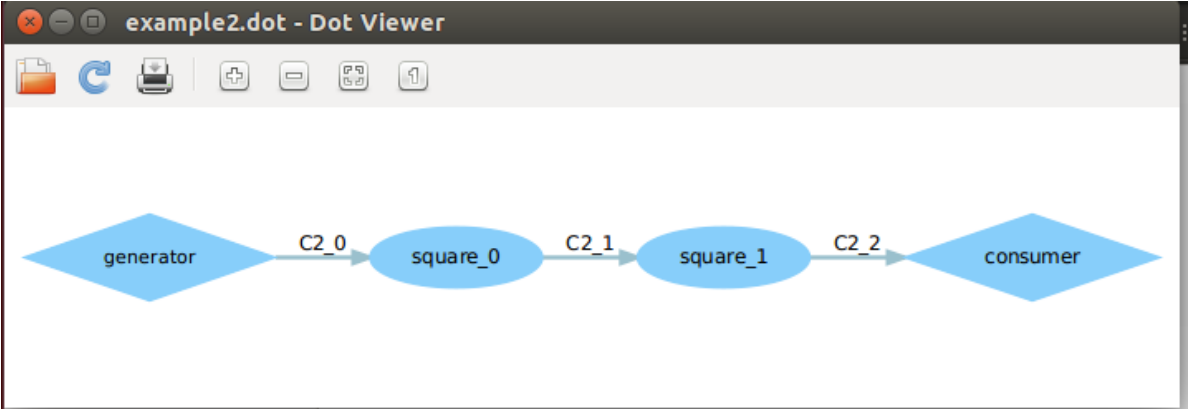
- **修改example2, 让3个square模块变成2个**

这个从xml文件就可以看出来最主要的就是直接修改

```
<variable value="3" name="N"/>  
为  
<variable value="2" name="N"/>
```

这个没什么好说的让他少迭代一次就能满足要求。重新build一下运行结果如下

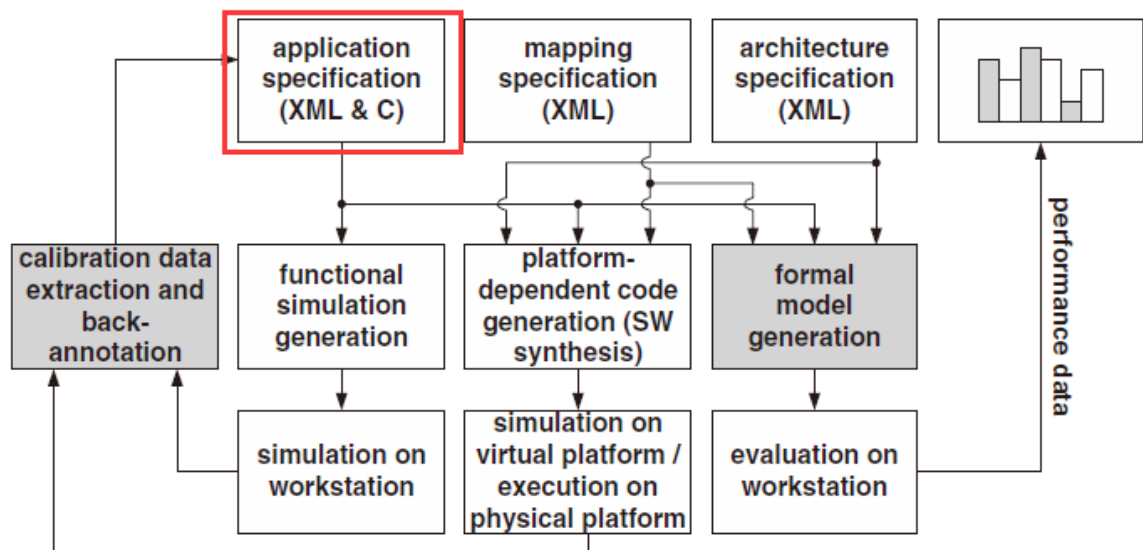
```
[echo] Run HdS application.  
[concat] consumer: 0.000000  
[concat] consumer: 1.000000  
[concat] consumer: 16.000000  
[concat] consumer: 81.000000  
[concat] consumer: 256.000000  
[concat] consumer: 625.000000  
[concat] consumer: 1296.000000  
[concat] consumer: 2401.000000  
[concat] consumer: 4096.000000  
[concat] consumer: 6561.000000  
[concat] consumer: 10000.000000  
[concat] consumer: 14641.000000  
[concat] consumer: 20736.000000  
[concat] consumer: 28561.000000  
[concat] consumer: 38416.000000  
[concat] consumer: 50625.000000  
[concat] consumer: 65536.000000  
[concat] consumer: 83521.000000  
[concat] consumer: 104976.000000  
[concat] consumer: 130321.000000  
  
BUILD SUCCESSFUL  
Total time: 9 seconds
```



3. 编译过程分析

DOL是一个平台独立的MPSoC编程环境，针对实时流和信号处理应用。它基于数据流程网络计算模型，并提供源到源代码生成器，以在不同的MPSoC平台上有效地执行DOL应用程序。数据流模型可以看作是一个协调模型，它允许将并行系统的编程视为两个不同活动的组合：其包含了操纵数多个过程和描述进程连接关系。指定数据流的应用程序，DOL使用两种不同的语言，C / C ++用来编程和XML来描述数据流过程网络的拓扑。选择这些语言是因为因为使用C / C ++允许重用现有的旧代码，XML易于处理，并且有大量可用工具。

对于DOL整个的设计过程来说我们的实验的修改只是应用程序编程的一部分。



整个编译的过程，个人认为实际上就是xml文件在控制.c的代码进行运行。一个xml控制多个.c文件，然后编译并多进程运行。编译器通过获取xml中的信息，加上xml中的代码进行联合编译，通过xml中的信息编译器可以知道哪个代码和哪个需要建立联系，也就是代码之间的关系。

```
sudo ant -f runexample.xml -Dnumber=1
```

指令输入后:

- 调用运行示例所需的所有目标
- 准备目录结构和复制源
- 验证和创建XML
- 运行DOL
- 创建并运行SystemC应用程序
- 运行XML检查器

五. 实验感想

这次实验修改实际上是非常简单的，都是改1个地方而。不过修改并不是最终重要的最重要的是如何理解这个编译过程。不过这个编译过程表面上看起来也挺容易理解，虽然说编译的时候具体做了什么不是很清楚。不过dol的代码写法和用法还是有了了解的。