# Modeling Kahn Process Networks in the Dynamic Dataflow Formalism

**Roy Kensmil**
**Leiden Institute of Advanced Computer Science (LIACS),**
**Leiden, The Netherlands**
**rkensmil@liacs.nl**

## ABSTRACT

In this paper an overview will be given of the processes and tools used to express Khan Process Networks generated by the Compaan Tool chain in the Dynamic Dataflow model of Ptolemy II.

**General Terms**
Embedded Systems Design, Computer Science

**Keywords**
Process Networks, Kahn Process Networks, Ptolemy II, Dynamic Data Flow, Synchronous Data Flow, Cloog, Threading, Scheduling

## 1. INTRODUCTION

Compaan [1] is an effort to automatically compile a subset of imperative programs into a concurrent representation. Compaan uses Matlab language as the imperative language and compiles programs in this language into a concurrent representation: a particular version of Process Networks, Kahn Process Networks [2].
Kahn process Networks that have been generated by the Compaan compiler need to be made accessible in such a way that the results can be analyzed by means of visualization and simulation. To perform simulation and analysis on the process network the Ptolemy II [3] framework is used.
The Ptolemy II is a software framework that can be used for modeling, simulation, and design of concurrent, real-time embedded systems. PtolemyII offers the possibility to simulate and visualize the process networks based on specifications according to a chosen computation model.

Compaan generates the network description in MoML, which is a modeling markup language based on XML used in Ptolemy II for specifying interconnections of parameterized components. The process generation step in this case, generates the Ptolemy II actors in the PN-domain. A MoML description can be executed as an application using a command-line interface or as a visual rendition in the Ptolemy II block diagram editor Vergil.

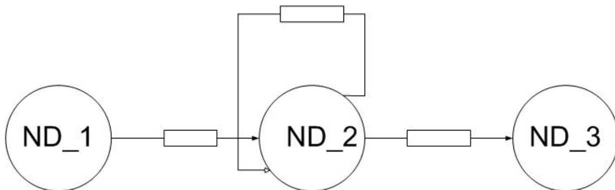


**Figure 1 :An Example of a simple network with 3 nodes**

In the MoML description of the network the following code is specified:

```
<entity name="ND_1" class="wild.ND_1">
    <property name="ehrhart" class="ptolemy.data.expr.Parameter" value=""></property>
    <port name="ND_1OP_1" class="ptolemy.actor.TypedIOPort"></port>
</entity>

<entity name="ND_2" class="wild.ND_2">
    <port name="ND_2IP_1" class="ptolemy.actor.TypedIOPort"></port>
    <port name="ND_2IP_2" class="ptolemy.actor.TypedIOPort"></port>
    <port name="ND_2OP_1" class="ptolemy.actor.TypedIOPort"></port>
    <port name="ND_2OP_1_d1" class="ptolemy.actor.TypedIOPort"></port>
</entity>

<entity name="ND_3" class="wild.ND_3">
    <port name="ND_3IP_3" class="ptolemy.actor.TypedIOPort"></port>
</entity>
```

In the generated java code the following functions for the actor ND_1 are specified in Figure 2:

```
public void initialize() throws IllegalActionException {
   super.initialize();
   boolean loaded = true;
   _returnValue = true;
}

/** fire the actor. */
public void fire() throws IllegalActionException
{
    System.out.println("START Node ND_1");
    for ( int j = (int) ceil(1) ; j <= (int) floor(1 ) ; j += 1 ) {
        //Init(out_0) ;
        // Variable: a_1(j)
        ND_1OP_1.broadcast( new DoubleToken(out_0) );
    } // for j
    System.out.println("FINISH Node ND_1");
}

/** post fire the actor. */
public boolean postfire() throws IllegalActionException {
    return false;
}
```

**Figure 2 Java code for a simple actor**

## 2. Problem Statement

Within the group the need has been expressed to expand the Compaan compiler with the ability to express the generated process networks [4] in the PtolemyII Dynamic dataflow formalism [5]. To understand the need a short summary will be presented with the differences between the KPN model and the DDF model. In the KPN model and DDF model each actor has its own input and output ports connected to FIFO channels. The main difference lies in the usage of threads. In the KPN model each actor is assigned to a separate thread. An actor in the KPN model can perform read or write operations. For a read operation to complete successfully the required data must be available in the neighboring channel of the input port. If the data is not available the actor will block on the read operation until data is available. For a write operation to complete successfully the channel of an output port must be able to accommodate data. If the data cannot be accommodated the actor will block until the channels have room to store data. A process network that uses this threading model has no control over the scheduler of the threads since the operating system is responsible for scheduling the threads. Using the KPN-model, the possibility exists that if one or more threads are blocked in a read or write operation the whole network can enter the deadlock state permanently. Figure 3 contains a graphical description of a process generated by the Compaan compiler.

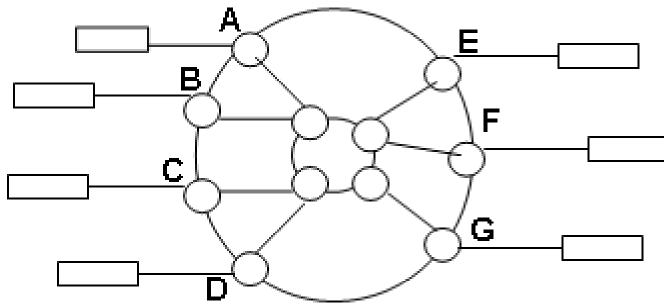


**Figure 3 graphical description of a process**

The input ports are:

A: referred to as ND_3IP_1_in_0 in Figure 7
B: referred to as ND_3IP_2_in_0 in Figure 7

C: referred to as ND_3IP_3_in_1 in Figure 7
D: referred to as ND_3IP_4_in_1 in Figure 7
The output ports are:

E: referred to as ND_3OP_1_d1_out_0 in Figure 7
F: referred to as ND_3OP_1_out_0 in Figure 7
G: referred to as ND_3OP_3_out_2 in Figure 7

The workings of an actor can be depicted as a table described in Figure 4

| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 |

**Figure 4 Firing sequences of the actor in Figure 3**

The value 0 in the table means that a port of an actor is in the inactive state and the value one means that the port of an actor is the active state. In the inactive state no data is read from its neighboring channel. If there is a one supplied in the column for an input port, the input port is in the active state. In the active state data is read from its channel.
One row of the table describes an execution cycle of an actor. A row of the table the displays the input ports and output ports that will be active during an execution cycle are described. Given that the execution of the actor will start in row one the actor will read data from the neighboring channels of input ports B and D and data will be output to the neighboring channels of ports F and G. In the second execution cycle (described in row2) data will be read from neighboring channels of input ports B and C and data will be output to neighboring channels of output ports F and G.

Figure 5 and Figure 6 contain a graphical description of the first an second execution cycle of the actor.

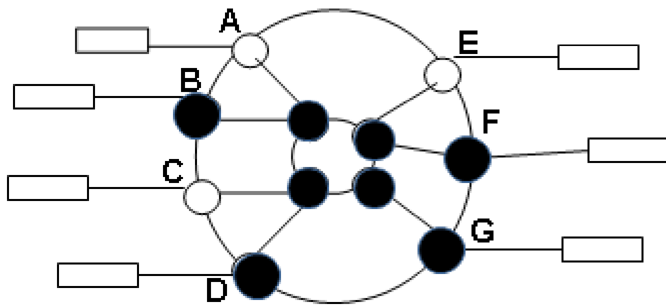


**Figure 5 firing during the fist execution cycle of the actor**

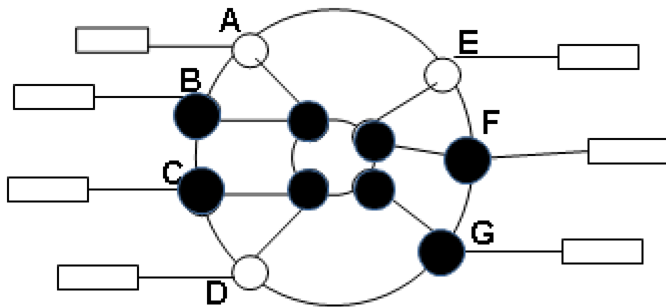


**Figure 6 firing during the second execution cycle of the actor**

In the current implementation of the Compaan the table necessary to specify an actor's execution is generated by programs written in Matlab. These programs can generate a text file that contains the patterns of zeros and ones describing the state of input and output ports during an execution cycle of an actor.

The text file for the actor described in Figure 3 has the following form:

ND_3IP_1_in_0_schedule: 20
0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
ND_3IP_2_in_0_schedule: 20
1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
ND_3IP_3_in_1_schedule: 20
0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1
ND_3IP_4_in_1_schedule: 20
1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0
ND_3OP_1_d1_out_0_schedule: 20
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1
ND_3OP_1_out_0_schedule: 20
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0
ND_3OP_3_out_2_schedule: 20
1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1,

**Figure 8 Initialization code for an actor in the PN domain**

The name ND_ is used as a prefix for the names of the output and input ports. The first four ports are input ports and the last 3 are output ports. The value 20, next to the name of each output ports defines the number of execution cycles the actor has to perform. An application making use of this text file must read the file in the following way:

- Input port ND_3IP_1_in_0_schedule has value zero meaning that it will be inactive
- Input port ND_3IP_2_in_0_schedule has value one meaning that it will be active
- Input port ND_3IP_3_in_1_schedule has value zero meaning that it will be inactive
- Input port ND_3IP_4_in_1_schedule has value one meaning that it will be active

- Output port ND_3OP_1_d1_out_0_schedule has value zero meaning that it will be inactive
- Output port ND_3OP_1_out_0_schedule has value one meaning that it will be active
- Output port ND_3OP_3_out_2_schedule has value one meaning that it will be active.

An application reads the text file column by column.

Each actor contains the initialize method. During the initialization of an actor in the process network the execution patterns generated using Matlab are read from the text file and loaded into system memory. The code for this process is given in Figure 8.

```
public void initialize() throws IllegalActionException {
    super.initialize();
    // Load the schedule program from file
    boolean loaded = loadProgram();
    N = ((IntToken) parameter_N.getToken()).intValue();
    K = ((IntToken) parameter_K.getToken()).intValue();
    _returnValue = true;
}
```

**Figure 8 Initialization code for an actor in the PN domain**

```
/** fire the actor. */
public void fire() throws IllegalActionException {
  for(int scheduleT = 0; scheduleT< 20; scheduleT++){
  System.out.println(" Node ND_3 value for scheduleT = " + scheduleT );
    if (ND_3IP_1_in_0_schedule[scheduleT] != 0) {
      in_0 = ( new Double(((DoubleToken)ND_3IP_1.get(0)).doubleValue()) ).intValue();
    }
    if (ND_3IP_2_in_0_schedule[scheduleT] != 0) {
      in_0 = ( new Double(((DoubleToken)ND_3IP_2.get(0)).doubleValue()) ).intValue();
    }
    if (ND_3IP_3_in_1_schedule[scheduleT] != 0) {
      in_1 = ( new Double(((DoubleToken)ND_3IP_3.get(0)).doubleValue()) ).intValue();
    }
    if (ND_3IP_4_in_1_schedule[scheduleT] != 0) {
      in_1 = ( new Double(((DoubleToken)ND_3IP_4.get(0)).doubleValue()) ).intValue();
    }
    // -- Function Call --
    // [out_0, out_1, out_2] = Vectorize(in_0, in_1);

    if (ND_3OP_1_out_0_schedule[scheduleT] != 0) {
      ND_3OP_1.broadcast( new DoubleToken(out_0) );
    }
    if (ND_3OP_1_d1_out_0_schedule[scheduleT] != 0) {
      ND_3OP_1_d1.broadcast( new DoubleToken(out_0) );
    }
    if (ND_3OP_3_out_2_schedule[scheduleT] != 0) {
      ND_3OP_3.broadcast( new DoubleToken(out_2) );
    }
  }
}
```

**Figure 9 code for the fire method of an actor of Figure 3 implemented in the PN domain**

In the fire method of the actor the read and write operation are performed. Based on the values generated in the Matlab file the actor will perform the pre specified number of iterations. Using the code sample specified in Figure 9 the problem of the blocking read/write can be demonstrated. The number 20 in the 'for loop' specifies the number of execution cycles for the actor. Notice the if statements specified for the input ports (ND_3IP_1_in_0_schedule, ND_3IP_3_in_1_schedule,

ND_3IP_3_in_1_schedule, ND_3IP_4_in_1_schedule). If the 'if statement' evaluates to true a read operation will be performed. But if there is no data available on the neighboring channel of the input port the operation of the actor will block at this point in the execution until the data is available. For the output ports, the same problem the problem of blocking can also occur. For the write ports there are also if statement specified. If a write port is active in an execution cycle and its channel is full, the write action will block until the channel has space. The post fire method is only used to determine if the actor has completed all executions.

```
/** post fire the actor. */
public boolean postfire() throws IllegalActionException {
     return false;
}
```

**Figure 10 code for the postfire method of an actor in the PN domain**

## 2.1 A strategy for implementing KPN expressed in the PtolemyII PN domain in the PtolemyII DDF domain

The Dynamic Dataflow (DDF) domain is a superset of the Synchronous Dataflow (SDF) and Boolean dataflow (BDF) domains. In the SDF domain, an actor consumes and produces a fixed number of tokens per firing. Hence scheduling is performed only at compile-time . In the DDF domain, an actor could change the production and consumption rates after each firing. The scheduler makes no attempt to construct a compile-time schedule; neither does it attempt to statically answer questions about deadlock and boundedness, which are fundamentally undecidable. Instead, each actor has a set of sequential firing rules (patterns) and can be fired if one of them is satisfied, i.e., one particular firing pattern forms a prefix of sequences of unconsumed tokens at input ports. The scheduler dynamically schedules the firing of actors according to some criteria.
From the information provided in the previous section, it is clear that the Kahn Process Networks heavily rely on the thread scheduling mechanism of an operating system. In DDF Domain the scheduling is done differently. As mentioned in the previous paragraph, an actor in the DDF domain can dynamically schedule production and consumption before each firing. Dynamically scheduling of the production and consumption rates provides the possibility to determine whether an input port of an actor can receive data or not and whether the output ports of an actor can send data or not. If it is known beforehand which actors should produce/consume data and which actors should not produce/consume data, the blocking reads and writes can be prevented. During an iteration of the network, all actors in the network that have the capability to produce or consume or both produce and consume data can be enabled to perform firings. Such a network does not require a separate thread for each actor. The scheduling of the actors that are able to perform operations without causing a read or write block can be done by one thread that functions like a scheduler.
In order to implement the actor executing in the PN domain as an actor running in the DDF domain, some of the properties of the actors in the PN domain needed to be investigated more thoroughly. Since both the PN and DDF are part of the Ptolemy II environment, the actors exhibit some of the same properties. In the DDF model, the execution of the actor is divided into three states. It is now also known that in the DDF domain it is possible to dynamically schedule consumption rates of an actor. Analyzing the file used for specifying the actors execution in the KPN domain, it can be concluded that actions specified in Figure 7 can be used a basis for specification of variable consumption rates during the execution of an actor.
The actor's execution in DDF can be divided into three states. These are the prefire, fire and postfire. The prefire state is initial state of the actor. In the prefire state conditions can be specified such as the ports that that will be enabled to read or write data. The fire state is the state in which the actors can send or receive data.
The postfire state is the state after the actor has completed the firing state. The postfire state is typically used for setting variables and 'cleaning up'. Actions necessary to be performed in the postfire state are setting values of the ports that will be disabled or enabled in the next iteration. In the postfire state it is also possible to indicate if the actor has completed all execution cycles in order to reset global variables. In the next section, a detailed overview will be given on how to express a sample application of a Kahn Process Network in the DDF formalism.

# 3. Solution approach

In order to Expanding the Compaan compiler to generate Kahn Process Networks expressed in the DDF model, the complex QRvr algorithm has been used. The QRvr algorithm [6] that has been provided for this project is a process network expressed in the KPN formalism and executable under the PtolemyII PN formalism. In the following subsections an overview will be given of:

1. the basic requirements for expressing a network in the KPN-formalism in the DDF formalism
2. Rewriting of a sample network in the KPN-formalism to a network expressed in the DDF formalism
3. Automating the process of conversion by making use of the ouput of Cloog.
4. Analyzing the output that Cloog [7] and automating the parameterization of the execution of the actors based on the output of Cloog.

## 3.1 Basic requirements for expressing a network in the KPN-formalism in the DDF formalism

In order to express a Process Network in the DDF formalism, a few basic steps must be undertaken:

1. Replace the for loop construction as specified in the actors running under the PN domain by a set of global variables which will control the execution.
2. Before an actor enters the fire state, the production and consumption rates of an actor must be set.

The changes proposed under point one can be realized by making use of the actors initialize, prefire and postfire methods. From the name of the initialize function it is clear that this function can be used for initializing global variables and other properties of the actor. In the case of an actor specified in the DDF domain the consumption pattern for the first cycle of the actor's fire method is set in the initialization method. The fire method of an actor specified in the DDF formalism should only determine which input ports and output ports should be activated based on the consumption patterns set in the previous iteration. As described earlier the post fire state of the actor is the state in which the actor enters after completing the firing state. The post fire method of an actor expressed in the DDF formalism method is the method that will be used to set the firing patterns (consumption/production rates) for the next firing cycle of an actor. Since the post fire state is the last state an actor can enter during its execution, the loop variable for keeping track of the number firing cycles to be performed can be incremented and a checked in this post fire state.

## 3.2 The QRvr example application

The code fragments in Figure 12, Figure 13 and Figure 14 show the code of the actor of the example QRvr application expressed in the Ptolemy II PN domain. Figure 11 shows the input file used for activation or deactivation of input ports.

ND_2OP_1_d1_out_0_schedule: 25
0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0,1, 1, 1, 1

ND_2OP_1_out_0_schedule: 25
1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,0, 0, 0, 0

**Figure 11 Input file used to activate or deactivate input and output ports. The number 25 specifies the number of iterations an actor has to perform**

```
public void initialize() throws IllegalActionException {
  super.initialize();
  // Load the schedule program from file
  boolean loaded = loadProgram();
  N = ((IntToken) parameter_N.getToken()).intValue();
  K = ((IntToken) parameter_K.getToken()).intValue();
  _returnValue = true;
}
```

**Figure 12 initialization code for an actor in the PtolemyII PN domain**

```
/** fire the actor. */
public void fire() throws IllegalActionException {


  for ( int T = 0; T < 25; T++) {
    System.out.println(" Node ND_2 value for T = " + scheduleT );

    // -- Function Call --
    // [out_0] = Read();
```

```
    if (ND_2OP_1_out_0_schedule[T] != 0) {
       ND_2OP_1.broadcast( new DoubleToken(out_0) );
    }
    if (ND_2OP_1_d1_out_0_schedule[T] != 0) {
       ND_2OP_1_d1.broadcast( new DoubleToken(out_0) );
    }
   }
   System.out.println("FINISH Node ND_2");
}
```

**Figure 13 code for the fire method of an actor in the PN domain**

```
/** post fire the actor. */
public boolean postfire() throws IllegalActionException {
  return false;
}
```

**Figure 14 code for the postfire method of an actor in the PN domain**

In this QRVR example application, the file generated by Matlab (figure11) is still used to determine which ports can read or write data during iteration. The main difference between the code for an actor in the DDF domain lies in the execution of the iterations of the actor. In the fire method, the only verification to be done is to determine which input port or output port is enabled. The code fragments from Figure 15 until Figure 24 show how the example QRvr application has been rewritten to execute in the PtolemyII data formalism.

```
public void initialize() throws IllegalActionException {
    super.initialize();
    // Load the schedule program from file
    patternCreator(scheduleT);
    _returnValue = true;
}
```

**Figure 15 code for the initialization method of an actor in PtolemyII DDF domain**

```
public boolean prefire() throws IllegalActionException {
       return true;
}
```

**Figure 16 code for the prefire method of an actor in the PtolemyII DDF domain**

```
  /** fire the actor. */
  public void fire() throws IllegalActionException {
    if(executionSetter()){
    System.out.println(" Node ND_2 value for scheduleT = " + scheduleT );

      if (testVariant1) {
        execVar1();
               if (_debugging) {
                       _debug("fire OP1");
               }

      }
      else if (testVariant2) {
        execVar2();
               if (_debugging) {
                       _debug("fire OP2");
               }
      }
    }
    System.out.println("FINISH Node ND_2");
  }
```

**Figure 17 code for the fire method of an actor in the PtolemyII DDF domain**

```
/** post fire the actor. */
public boolean postfire() throws IllegalActionException {

    if(testVariant1){
        testVariant2 = true;
         testVariant1 = false;
    }
    else if(testVariant2){
```

```
        V2_j++;
        if(V2_j > loopMaxV2){
             V2_j = 2;
             V1_i++;
             if(V1_i > loopMaxV1){
                 testVariant2 = false;
                 endCheck = true;
             }
             else{
                 testVariant2 = false;
                 testVariant1 = true;
             }
        }
    }

    scheduleT++;
    if (_debugging) {
        _debug(" schedCheck "+ scheduleT);
    }

    if(executionSetter()){
        patternCreator(scheduleT);
          return true;
    }
    if (_debugging) {
        _debug(" Actor wrapup ");
    }
    return false;
}
```

**Figure 18 code for the postfire method of an actor in the PtolemyII DDF domain**

In the postfire method of the actors, the loop variable is incremented and the verification is done to determine if the actor must stop executing.

```
public void patternCreator(int T_In) throws IllegalActionException {

    if(testVariant1){
        setVariant1();
    }
    else if(testVariant2){
        setVariant2();
    }

    if (_debugging) {
        _debug("Pattern creator called");
    }
}
```

**Figure 19 The PatternCreator function**

```
public boolean executionSetter(){

    if(!endCheck){
        return true;
    }
    return false;
}
```

**Figure 20 The ExecutionSetter function. Determines whether all the firing cycles of the actor have completed**

```
public void setVariant1() throws IllegalActionException{
    ND_2OP_1_outputPat0= new IntToken(1);
    ND_2OP_1_tokenConsumptionRate.setToken( ND_2OP_1_outputPat0);

    ND_2OP_1_d1_outputPat1= new IntToken(0);
    ND_2OP_1_d1_tokenConsumptionRate.setToken(ND_2OP_1_d1_outputPat1);
}
```

**Figure 21The setVariant1() function. A function for setting production and consumption rates**

```
public void setVariant2() throws IllegalActionException{
    ND_2OP_1_outputPat0= new IntToken(0);
    ND_2OP_1_tokenConsumptionRate.setToken(ND_2OP_1_outputPat0);

    ND_2OP_1_d1_outputPat1= new IntToken(1);
    ND_2OP_1_d1_tokenConsumptionRate.setToken(ND_2OP_1_d1_outputPat1);
}
```

**Figure 22 The setVariant2() function. A function for setting production and consumption rates**

```
public void execVar1() throws IllegalActionException{
    ND_2OP_1.broadcast( new DoubleToken(out_0) );
}
```

**Figure 23 The ExecutionSetter function.  Specifies output of data to a port**

```
public void execVar2() throws IllegalActionException {
    ND_2OP_1_d1.broadcast( new DoubleToken(out_0) );
}
```

**Figure 24 The ExecutionSetter function. Specifies output of data to a port**

Notice the difference between the firing methods and the post fire methods of the PN and the DDF model. In the PN model, the firing method contains a for loop. Once the firing function is entered it will be executed for a fixed number of iterations. The ‘if statements’ used in the ‘for loop’ of the firing method of the PN Domain are the main cause of the blocking read/write problem mentioned in the problem statement. During each iteration the input and output ports for an actor will be set in an inactive or active state as can be seen seen in the ‘if statements’. But there is no mechanism built in to check whether the channel of an input or output port contains data or can receive data. The firing method of the actor expressed in the DDF Domain does not contain a for loop. The number of iterations to be performed is now controlled by a global variable (scheduleT) which is updated in the postfire method seen in Figure 18. The global variable is initialized to zero at the start of the actor’s execution. In the firing method, the iteration variable is not used anymore. In the fire method only the activation of input and or output ports occur, based on the consumption rates set in the previous iteration of the actor. Figure 17 shows the fire method only making use of two Boolean variables testvar1 and testvar2 for activation of input or output ports. The postfire method contains the code for incrementing the variable for the number of iterations the actor should perform and the method for setting the consumption and production rates for the next iteration the actor. In the postfire method of the actor seen in Figure 18, a call to another method is specified, named the patternCreator. The patternCreator method is used to set the consumption and production rates (firings) for the actors in the DDF model. Before an actor enters the firing state, it is possible to specify if the actor will be sending or receiving data or perform both actions. Having this information available, it can also be determined which channels of an input port contains data and which channels output port can receive data. When the actor enters its firing state the read and write operations will not be blocked due to empty channels or full channels.

## 3.3 Analyzing the output that Cloog generated for the matrices generated in  Matlab

CLoog [7] is free software and library generating loops for scanning Z-polyhedra. CLoog has been originally written to solve the code generation problem for optimizing compilers based on the polytope model. Using an input matrix, such as the matrix in Figure 25 specified for an actor in KPN, Cloog generates a series of loops and nested loop. In the loop structures for a KPN actor the following statements and variables are generated:

- Variables M and N
- if conditions for the variables M and N
- for loops
- nested for loop
- Variables for the iterations of loops(i,j,k )
- Statements S(1)...S(n) with or without parameters.

The values of variables M en N determine which ‘for loop’ statements can be executed by an actor. Using the values of variables M en N the if statements can determine which loop statements will be executed during execution of the code.
The statements that will be executed in the loop statements can be mapped one to one to the input ports and output ports of an

actor in a process network.
The above implies that the statically generated patterns can be replaced by the values which are dynamically generated by Cloog.

```
# language: C
c

# parameters
4 4
1  -1  0  16
1  1  0  -1
1  0  -1  1000
1  0  1  -1
0


2

1
5 6
1  1  0  0  0  -1
1  -1  0  0  1  0
1  0  1  0  0  -1
1  0  -1  1  0  0
0  0  1  0  0  -1
0  0  0

1
5 6
1  1  0  0  0  -1
1  -1  0  0  1  0
1  0  1  0  0  -1
1  0  -1  1  0  0
1  0  1  0  0  -2
0  0  0
0

0
```

**Figure 25 An example input matrix for Cloog for the actor describe in section 3.2**

```
/* Generated from ..\\test\ND_2.cloog by CLooG v0.14 12/07/07 bits in 1.00s. */
if (M >= 2) {
  for (i=1;i<=N;i++) {
    S1(j = 1) ;
    for (j=2;j<=M;j++) {
      S2 ;
    }
  }
}
if (M == 1) {
  for (i=1;i<=N;i++) {
    S1(j = 1) ;
}
```

**Figure 26 Output generated by Cloog based on the input matrix seen in Figure 25. This code will be used for the parameterization of the actor specified in section 3.2**

As can be seen from Figure 26 The value M in the code fragment determines which output ports of an actor will be active during the iterations of an actor. If the value of variable M equals 2, output ports 1 and 2, specified as S1 and S2 in Figure 26, of the actor will be active during the iteration of an actor. If the value of variable M equals 1, output ports 1, specified as S1 and in Figure 26, of the actor will be active during the iteration of an actor.

## 3.4 Automating the parameterization of the execution of the actors

In the previous section it has been proven that it is possible to convert Kahn Process Networks processes expressed in the Ptolemy II PN Domain to Kahn Process Networks processes expressed in the Ptolemy II DDF Domain. But the solution suggested in section 3.3 requires that the process of conversion must be performed by hand. Automating the process of the conversion requires a ‘bridge’, that can translate the code generated by Matlab and Cloog to Java code that can be processed

in the Java code generated by the Compaan compiler. The decision was made to use the Java Native Interface. The Java Native Interface (JNI) is the native programming interface for Java that is part of the JDK. The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. In addition, the Invocation API allows programmers to embed the Java Virtual Machine into native applications. Programmers use the JNI to write native methods to handle those situations when an application cannot be written entirely in the Java programming language.
To capture the structure of the code generated by Cloog, some classes have been created to represent the various data structures generated by Cloog. Such elements include:

- a class for representing if statements,
- a class for representing for-loop statements
- a class for representing assignments that can occur within a 'for' or 'if statement'.

The list of statements executed within an 'if-statement' or 'for loop' are included as strings in the respective classes. In the Compaan compiler a Tree data structure (a ParseTree) has been implemented to represent various constructs generated by a program. To represent the structure of code generated by Cloog the C program mentioned above uses the JNI to invoke the existing methods of the ParseTree data structure created in Compaan. Figure gives a description of the classes and the relationships between the classes developed in Java to represent the data structures generated by Cloog. At the top of the hierarchy is the Cloog Constuct. The Cloog Construct serves as a template for every new data structure to be created for the Parse Tree of the Compaan Compiler. Figure 28 and Figure 29 give an overview of the diagrams for modeling an 'if statement using' theory from compiler construction. Figures 30 and 31 give an overview of how the if and for constructs have been modeled based on the ParseTree datastructure of Compaan. Using the code fragment from Figure 26 an overview will be given of the new data structures created for the Parse Tree of the Compaan Compiler. The guards of the if construct, as seen in figure 30, contains two variables and an operator. Using Figure 26 the guard can be associated with the expressions M >= 2 and M == 1. The Variant data structure consist of at least one VariantString. A VariantString data structure can be associated with the statement S1 and S2 of the code in Figure 26. As mentioned at the end of section 3.3 statement S1 and S2 can be associated with the output ports of the actor. Thus the Variant data structure represents the output and input ports that will be active if the condition specified in the Guard of the 'if statement' is satisfied. In the case of the for statement, as depicted in Figure 31, the expression determines the number of iterations that the variant(collection of input and output ports) will be active. If for example in Figure 26 the condition M == 1 is satisfied the variant containing S1 would be active for iterations 1 to N. Figure 32 depicts a diagram for representing an actor expressed in the DDF formalism in the Compaan Compiler.

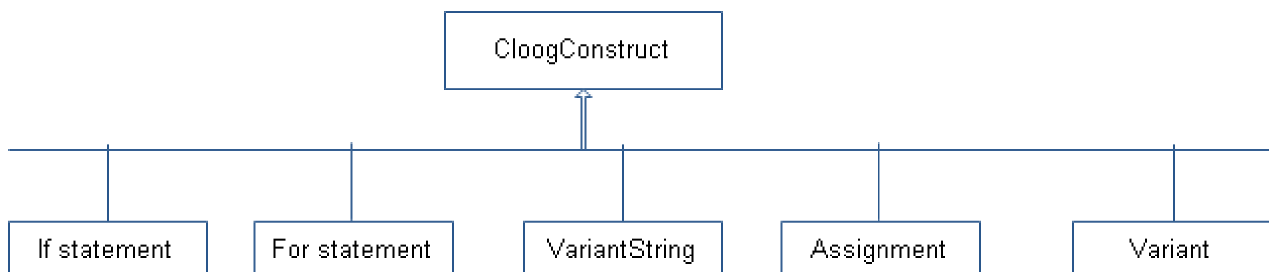


**Figure 27 Diagram of the Java classes**

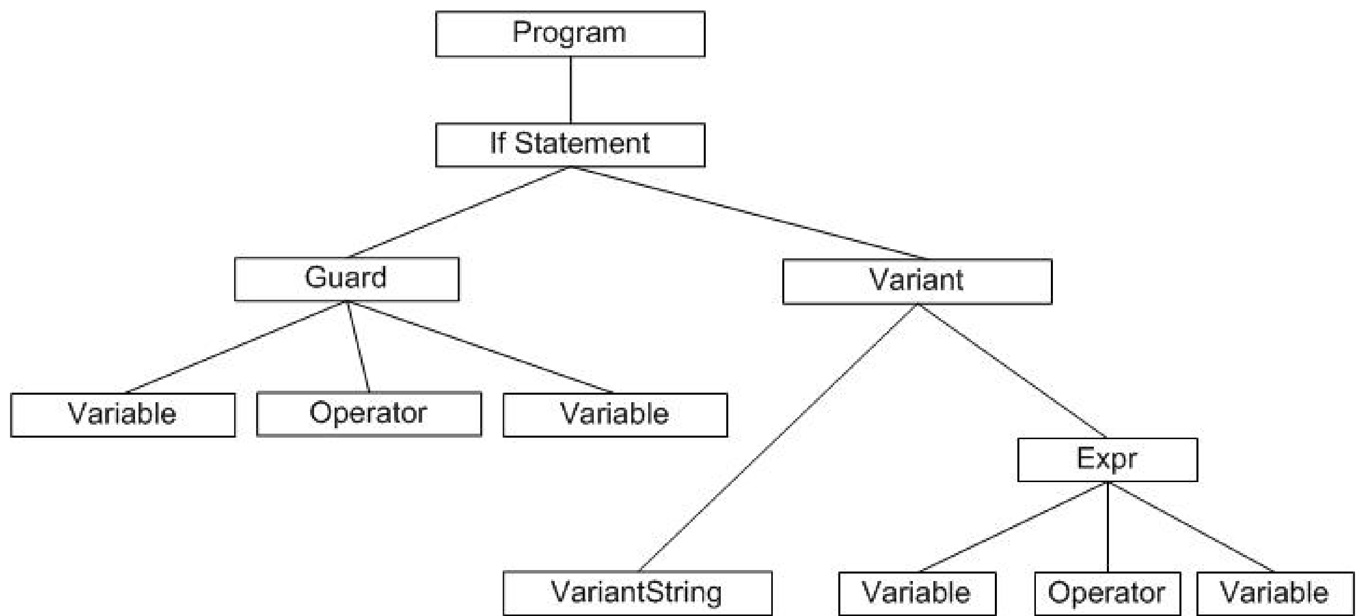


**Figure 28 If construct**

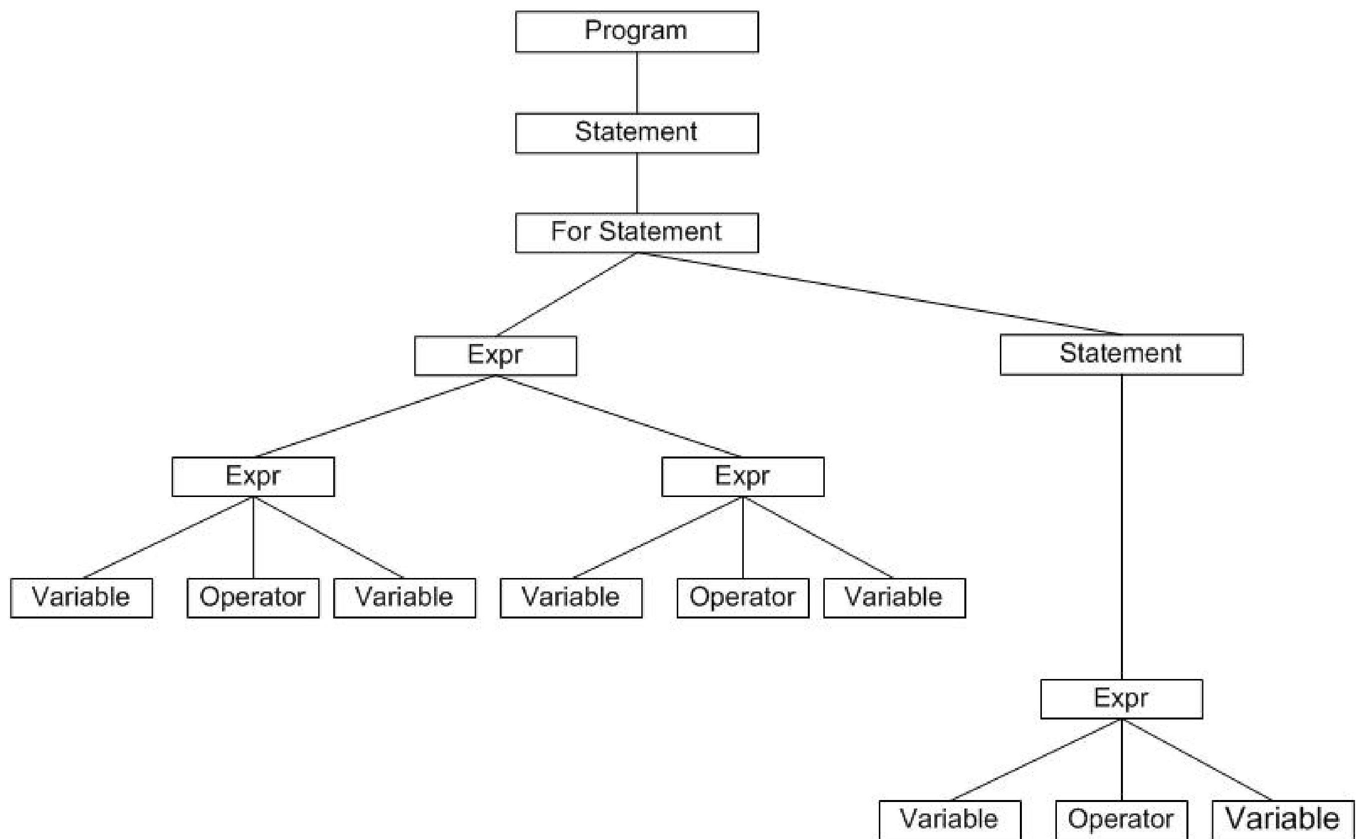


**Figure 29 For construct**

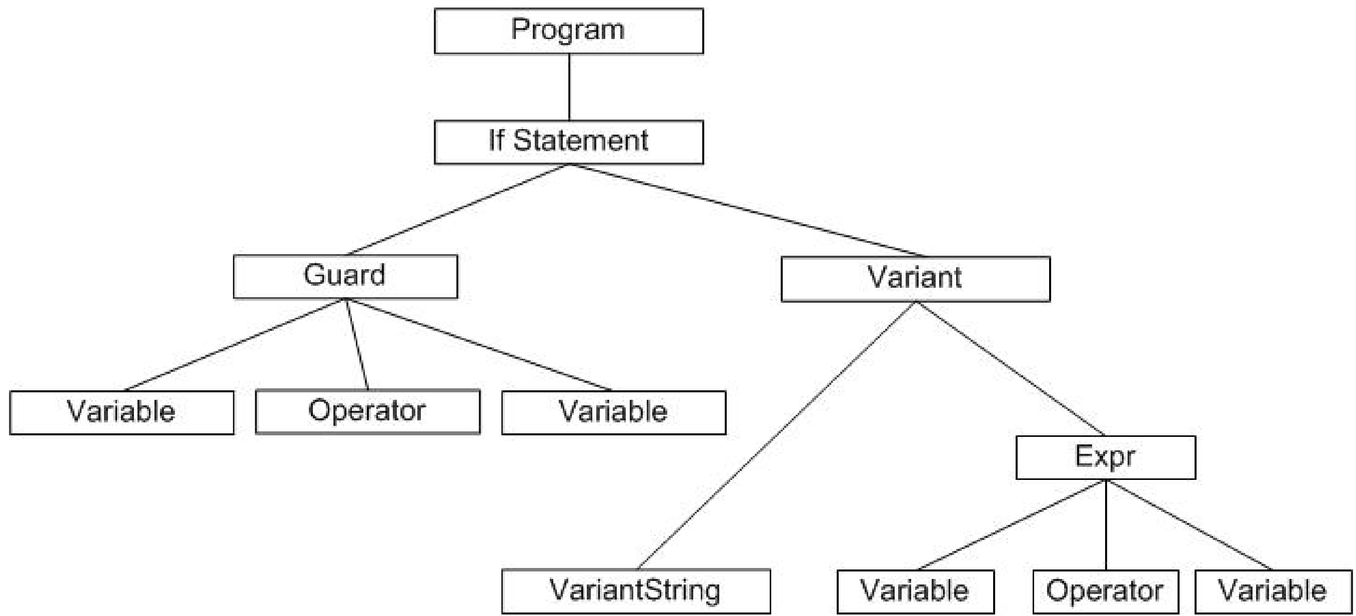


**Figure 30 If Construct represented in Compaan**

Program
For Statement
Expr
Expr
Expr
Variable
Operator
Variable
Variable
Operator
Variable
Variant
VariantString
Expr
Variable
Operator
Variable

**Figure 31 For Construct represented in Compaan**

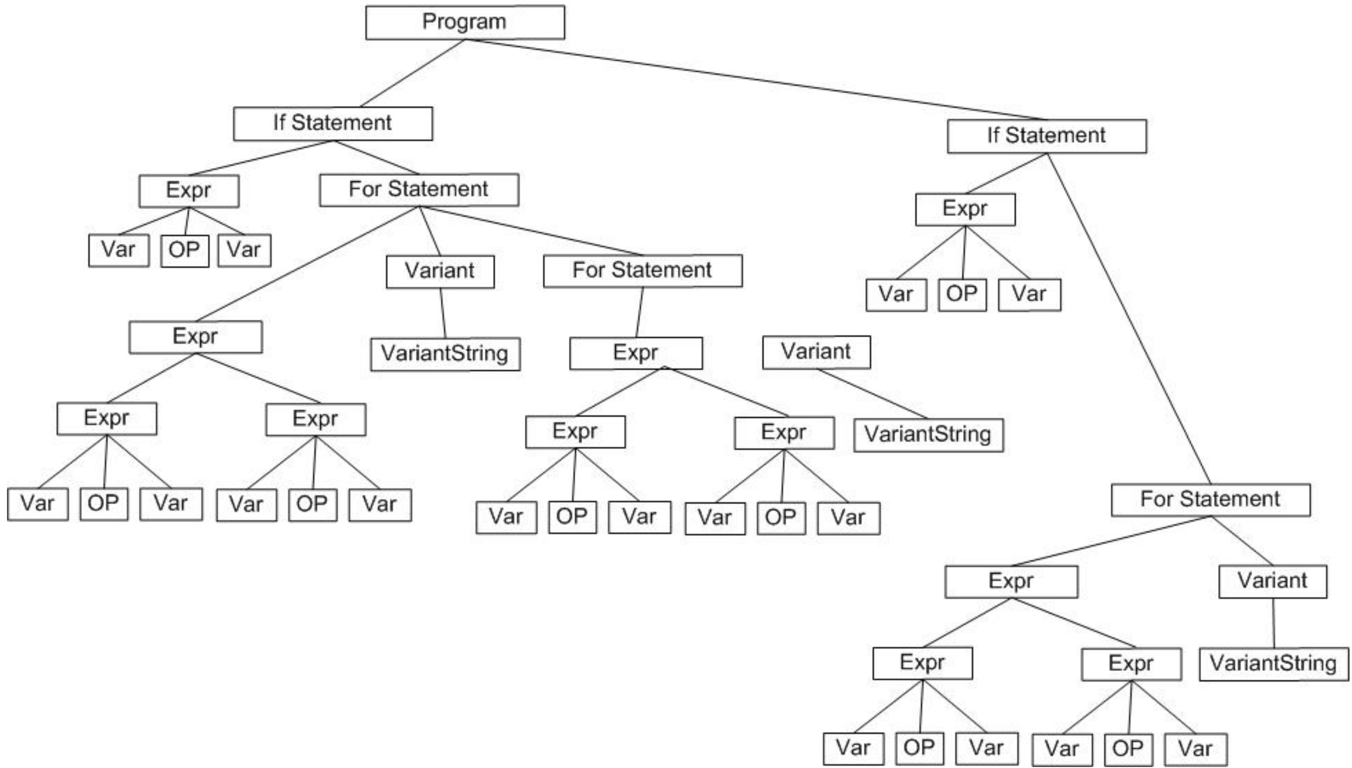


**Figure 32 the Tree structure for the example program**

## 4. Conclusion

Through investigation and experiments it has been proven that the Process Networks generated by the Compaan compiler for the Ptolemy II Process Network Domain can be expressed in the PtolemyII Dynamic Dataflow Domain. The main problem to be solved in the conversion process was the removal of the blocking read/write operation caused by the structure of the 'for loops' in the Kahn Process Networks generated by the Compaan compiler. Having recognized the capability of Cloog to generate optimal code for the execution 'for loops', the decision was made to apply the data structures generated by Cloog to express the Process Networks generated by Compaan in the PtolemyII Dynamic Dataflow formalism. The code generated by Cloog has the same execution patterns as the code of processes expressed in the PtolemyII Dynamic Dataflow formalism.
To enable Compaan to automatically generate a process network expressed in the PtolemyII Dynamic Dataflow formalism it is necessary to convert the data structures generated by Cloog into data structures which can be used in Java. The conversion of the data structures generated by Cloog is done by making use of the Java Native Interface. By making use of a dynamic link library (.dll) the data contained in the Cloog data structures are passed on to the Java data structures of the Compaan Compiler.

## ACKNOWLEDGEMENTS

I would hereby like to thank Bart Kienhuis for his support during the development phase of the project and for the support given during the writing of this paper.

# REFERENCES

[1] The Compaan Project: http://www.liacs.nl/~cserc/compaan/

[2] Kahn Process Networks: http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/DATE_04.pdf

[3] The Berkeley PtolemyII Framework: http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm

[4] The Process Network (PN) Domain of the PtolemyII Framework:
http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIlatest/ptII/ptolemy/domains/pn/doc/body.htm

[5] The Dynamic Dataflow (DDF) Domain of the PtolemyII Framework:
http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIlatest/ptII/ptolemy/domains/ddf/doc/body.htm

[6] QRvr matrix decomposition algorithm
R.L. Walke, R.W.M. Smith, and G. Lightbody. 20GFLOPS QR processor on a Xilinx Virtex-e FPGA. In proceedings of SPIE advanced signal, 1999

[7] Cedric Bastoul, A Loop Generator For Scanning Polyhedra
Edition 2.1, for CLooG 0.14.1, October 15th 2007
http://www.bastoul.net/cloog/documentation.php