Building Logistic Games in CPN Tools

Internship report

Marcel Welters (460859)
M.H.E.Welters@student.tue.nl
Technische Universiteit Eindhoven

Eindhoven, January 22nd 2005

Supervisors: Prof. Dr. ir. W.M.P. v.d. Aalst and Dr. M. Voorhoeve

Table of contents

1. Introduction

This is the final report of the project "*Logistic games in CPN Tools*". The project started with searching for logistic games and modeling them using colored Petri Nets. There are a lot of logistic games available on the internet. However most of them are about the financial part of logistics, instead of the operational part. The following games are modeled in CPN Tools.

The *Mortgage Game* is a service oriented supply chain, with a make-to-order environment. The goal of this game is to minimize personnel (capacity) cost. You control one part of the mortgage processing, for instance the requesting part. You have to decide how much capacity is necessary, based on the backlog of mortgage requests. In this game this is called setting a target capacity. You can adjust capacity by hiring or firing people. However it costs time and money to adjust capacity. You need time to hire new employees, to increase capacity. Further you have to give notice to employees if you want to fire them. The people that work for you try to process as much requests as possible. Processed requests are put in the backlog one link downstream. Assume that every mortgage request is processed eventually. Further every part of the chain is another company, and they don't share information about the number of request each one handled.

The *Computerized Kanban Game* takes place in a Kanban production environment. The goal of this game is to meet demand, while staying within the capacity bound. The player with minimal costs in the end wins. You control a work cell or production unit. It contains one production line that is capable of producing four products. For each period the sales per product of that period are known. Further the stock level of each of the four products (A,B,C,D) is also known. You have to decide how much to produce of each product each period.

In the *Distribution Game* you own and control a small distribution network. It is a two level distribution situation. You control the quantity that is shipped from the manufacturer to the warehouse. Further you control the shipment quantities to your three retailers. The goal of this game is to cost effectively manage the flow of goods to supply random customer demand at multiple locations. Whoever gets the highest net profit wins. If the shops don't have enough inventory, they can't fulfill demand and you miss profits. Backlog and backorders are not possible in this game; but backorders can be easily added.

The *Beer Game* is a supply chain with a make-to-stock environment. The goal of this game is to minimize costs throughout the entire chain, or you can play with another goal, minimizing your own costs. Shortages are penalized more than stocking. In the Beer Game you control one stock point and you have to decide how many barrels of beer you're going to order.

After modeling these games in CPN Tools, Java applications that can communicate with each of the CPN models were made. The communication between the CPN model and the java application was done using the Comms/CPN library. This library provides functions for TCP/IP communication with an external java application in standard ML. The Java/CPN library is the java peer module of Comms/CPN, providing functions for communication with CPN Tools over TCP/IP in Java.

A website with the logistic games together with a short manual also has to be delivered. The website has to contain at least the CPN models, the applications, the source code and a short user manual.

So the project consists of three parts:

- Searching for and modeling of the logistic games in CPN Tools
- Turning the models into games by writing a Java application for each CPN model
- Creating a website that contains the logistic games

First the setup of the communication between the Java application and CPN Tools is described. Next the Java applications are discussed. After these chapters, the modeling of the games is discussed, followed by a conclusion. In the appendices you can find the following: how to make a simple simulation model in CPN Tools, simulation results of the Beer Game, a short manual how to start the games. Although the topics of the appendices helped a lot in the project, they are not included in the main text. This is because the topics differ too much from the actual project.

In this report an overview is presented of how the entire project is made. Modeling and implementation issues are discussed. For detailed description refer to the CPN models and the Java source code. Not all sub pages are described in this report. This report is a good starting point for those who want understand the CPN models and the Java applications.

When describing the CPN models, the transition and place names are written in italics.

For a description of how the user interfaces work, refer to the manuals for the games.

2. Designing the games

Colored Petri Nets, when modeled in CPN Tools are restricted in their ability to interact with external processes, such as java applications. Extending CPN Tools by providing a communications infrastructure allows communication to be established between CPN models and external processes [13, 1]. The Comms/CPN library has been developed to extend CPN Tools with such external communication facilities. Originally the Comms/CPN library was developed for Design/CPN, the predecessor of CPN Tools (it is included in CPN Tools version 1.1.0 and higher). The motivation for developing such external communication facilities comes from the desire to visualize the simulation of a CPN model. Currently in CPN Tools the only form of visualization is the token game. Sometimes it would be beneficial to visualize the system using an external visualization package. This can be useful for people that aren't familiar with CPN models that are interested in for instance the system behavior.

The external visualization package used in this project is the Chart2D visualization package [14]. This is a Java class library for visualizing data using two-dimensional graphical objects. The library provides charts such as pie charts and graph charts. To be able to use this package requires some knowledge of Java and Swing in particular.

2.1. Comms/CPN

The Comms/CPN library has been developed to allow communication between CPN Tools and external processes via TCP/IP. The library can be used to provide CPN models with their own GUI. All the logic is in the CPN model, while the user interface handles the user interaction and visualization. For the design overview and the requirements refer to [13,2]. Here the protocol used by Comms/CPN is described in detail.

Next the Comms/CPN functions that were used in this project are described.

- `ConnManagementLayer.acceptConnection: string * int -> unit`
  Provides server behavior, and allows external processes to connect to CPN Tools. The first argument must be a unique string identifier to be associated with the connection. The second argument is a port number. The function checks that the string identifier is unique, and then listens on the given port for incoming connection requests. This causes CPN Tools to block until an incoming connection request is received. When this happens, a connection is established with the external process requesting the connection.

- `ConnManagementLayer.send: string * 'a * ('a -> Word*Vector.vector) -> unit`
  Allows users to send any type of data to external processes. The function is polymorphic, in the sense that the data passed to it for sending can be of any type, including user defined types. Three parameters are passed to this function as input. The first is a string identifier for the connection, the second is the data to send, and the third is a function to encode the data to send. The purpose of the encoding function is to encode the data to send into a sequence of bytes. This allows the data to be of any type, provided an encoding function exists for that type. The send function retrieves the connection corresponding to the given string identifier. The return type of this function is type unit.

- `ConnManagementLayer.receive: string * (Word8Vector.vector -> 'a) -> 'a`
  Allows users to receive any type of data from an external process. The receive function is polymorphic in the same way as the ConnManagementLayer.send

function. The parameters to this function are a string identifier for a connection and a decoding function, to decode the received byte vector into the appropriate data type. The resulting decoded data is returned.

- `ConnManagementLayer.closeConnection: string -> unit`
  Allows users to close a connection. The string identifier of the connection to be closed is passed to this function as the argument. A search of the connections is conducted to ensure that a connection exists with that string identifier. If the connection does not exist, an ElementMissingExn exception is raised.

The Comms/CPN library provides the following encode and decode functions:

- `stringEncode s` converts string s to Word8Vector.vector
- `stringDecode v` converts Word8Vector.vector v to a string
- `integerEncode i` converts integer i to a Word8Vector.vector
- `integerDecode v` converts Word8Vector.vector v to an integer

It is also possible to write your own encode and decode functions. In this project this was not necessary because only integer and string values are used in the communication.

2.2. Java/CPN

Java/CPN [13,4.1] allows Java processes to communicate with CPN Tools through Comms/CPN. The current implementation of Java/CPN is the minimal implementation necessary to enable communication. Java/CPN incorporates the same functionality of the messaging and communication layers from Comms/CPN. The communication layer functionality from Comms/CPN as well as TCP/IP is already encapsulated in the socket object provided by Java through the use of Socket methods and the input and output streams available from the socket itself. There is no connection management implemented in Java/CPN, however it implements the same protocol as the messaging layer from Comms/CPN.

Next the Java/CPN methods used in this project are described:

- The connect method [13,4.1] takes a hostname and port number as arguments, and attempts to establish a connection as a client to the given port on the given host. This method does not return a value. Once the connection has been established input and output streams are extracted from the socket to enable the transmission and reception of bytes.

- The send method [13,4.1] takes a ByteArrayInputStream object (a Java object for holding sequences of bytes, acting as input) as the argument. The data packets formed are transmitted to the external process (in this case CPN Tools) through methods acting on the output stream of the socket. The send method does not return a value.

- The receive method [13,4.1] has no arguments. It uses methods that act on the input stream of the socket to firstly receive a header byte, and then receive the number of payload bytes specified in the header from the external process (also in this case CPN Tools). The payload bytes are stored in a ByteArrayOutputStream object (a Java object for storing bytes as output) as each segment of payload data is received. The receive method returns the ByteArrayOutputStream object.

- The disconnect method [13,4.1] has no argument and returns no value. It acts in the same way as the disconnect function from the Comms/CPN library, except that it also closes the input and output streams from the socket before the socket itself is closed.

Methods external to the Java/CPN class must be used to convert from data (i.e. string, integer) into a ByteArrayInputStream object, and from a ByteArrayOutputStream object back into data. This similar to the encoding and decoding functions passed into the send and receive functions of the Connection Management Layer in Comms/CPN. These methods are implemented for strings in the Encode Decode Java class.

2.3. How to combine everything

In the figure below is shown how the different are connected to each other. CPN Tools GUI is the CPN Tools program. The standard ML (SML) process runs in the background and is the simulation engine of CPN Tools. CPN Tools communicates with the simulation engine through TCP/IP. The Java application visualizes the simulator state. As stated above this is done with Comms/CPN and Java/CPN, which also communicate with each other over TCP/IP.

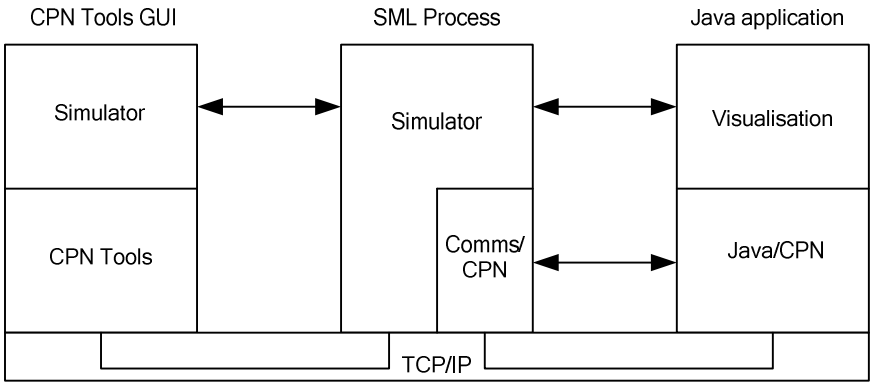

Figure 1a - Context of Comms/CPN and Java/CPN

Functions from the Comms/CPN library can be added to a CPN model by putting these functions on the appropriate arcs or in code segments of transitions. The approach taken in this project was to first create CPN models for the games. After this the Comms/CPN functions were added to transition code segments. How it is done for each game is described in the next four chapters.

The Java/CPN library consists of three classes: JavaCPN, JavaCPNInterface and EncodeDecode. To add Java/CPN support to your Java application, these three files need to be included in your application. When they are properly included, the communication can be realized as follows. Define a variable of type JavaCPN, say myVariable.

To connect your application to CPN Tools you call the connect method of myVariable (with the correct arguments). The Java application now tries to connect to CPN Tools. To disconnect your application from CPN Tools, you call the disconnect method of myVariable.

Sending data to CPN Tools requires you to encode that data into a ByteArrayInputStream object. For this you need the Encode Decode class. This class provides methods for encoding your data.

Receiving data from CPN requires you to decode that received data into a ByteArrayOutputStream object. Here you also need the EncodeDecode class, which provides methods for decoding the data.

For the implementation details refer to the source code of the applications. Screenshots of the applications can be found in appendix A5.

As a starting point for the implementation of the Java application [16] was used. This website contains a CPN model of the dining philosophers with external communication.

2.4. Cost meter CPN model

In the games costs have to be calculated. Modeling order costs is straightforward. Everytime an order is placed, add the order costs to the costs already made. However calculating inventory holding cost and backorder costs were not trivial in modeling. Calculating the costs can be done using the cost meter in figure 1b.

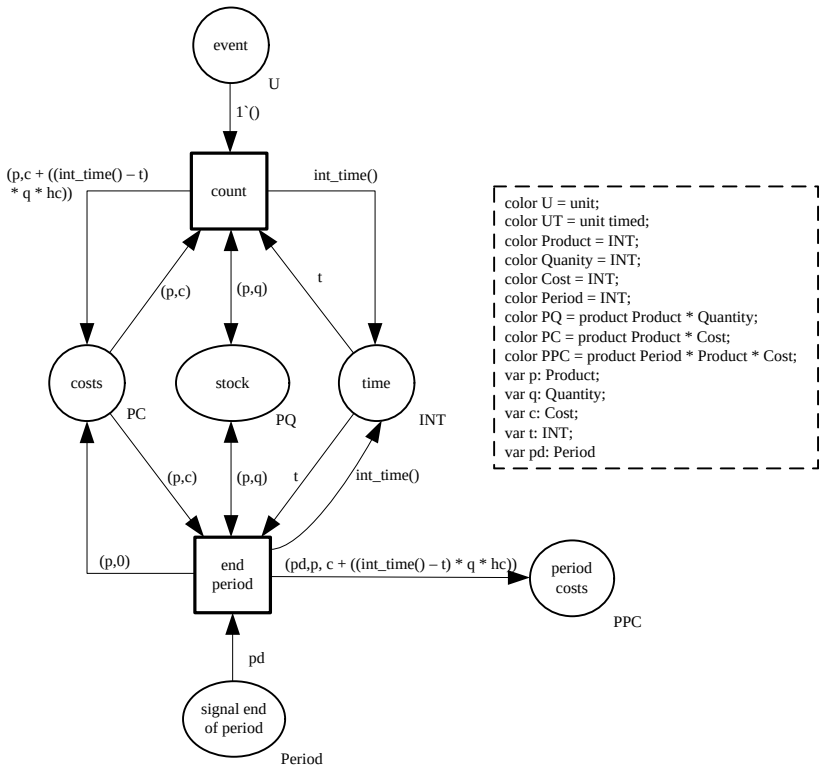

Figure 1b - Cost meter CPN model

The cost meter updates the cost when an event occurs. In this case this can be a replenishment of the stock or an order delivery. These events change the stock position. A change in the stock position means that the holding costs have changed. Occurrence of an event is modeled with place *event;* a token is produced when an event occurred. To calculate the inventory holding costs, we use that we know the stock level and the costs until the last event occurred, and we know when the new event occurs. This gives a time interval for which we can calculate the cost and add them to costs that were incurred already. The time of the last event is stored in place *time*, the costs in place *cost* and the stock position in place *stock*. Now multiple events can happen during a period. Suppose we defined a period being 100 time units in CPN Tools. Within these 100 time units events can occur and transition *count* will keep updating the costs. When the time reaches 100, *end period* will fire, resetting the costs and the time of the last event. The costs for that period are put in place period costs. Transition *count* will start with all costs set to zero. *End period* will fire again at time 200, etc.

This was generalized to the following in CPN ML:

```
fun costs(cpu,q,c,t)= c + (cpu * q * (int_time()-t));
```

- cpu are the costs per unit per period
- q is the quantity
- c are the costs already incurred until time t
- t is the time until which q and c are updated
- int_time() is the current time

(int_time() – t) is the time interval between two events. Multiplying q with cpu gives the total costs for quantity q per period. Since the time interval is known, we can calculate the total costs incurred and add them to c.

This cost meter is used in the *Mortgage Game*, the *Distribution Game* and the *Beer Game*.

3. The Mortgage Game

3.1. Description

The mortgage service game models a service supply chain. A service supply chain behaves differently from finished goods inventory distribution chains. Service supply chains typically do not have inventory stocks that can be replenished by placing orders. They have backlogs that are managed indirectly through service capacity adjustments. The Beer game models a distribution chain very well, but it can not be used for a service oriented supply chain. For reasons of simplicity the Beer game assumes infinite manufacturing capacity. This assumption works well in some distribution supply chains, however in a service supply chain this does not hold. In a service supply chain the capacity needs to be adjusted. The authors of [1] have developed a game that more closely resembles a service supply chain and the types of decisions that have to be made. The game they have developed has three main features:

1. The supply chain has multiple stages, performing a sequence of operations
2. Finished goods inventory is not possible, because the product is a service. The service supply chain is a make-to-order environment
3. Each stage manages the order backlog only by adjusting capacity

The mortgage game models or represents a simplified mortgage approval process. The idea behind the model is to look at some aspects of a service oriented supply chain and not to model the process with detail. Also complexities that are typical for the mortgage business are not modeled.

This model was originally implemented in a program called VensimPLE. This is a simulation package made by a company called Ventana [2]. The VensimPLE program can be used to analyze system dynamics. This has some complications for the modeling of the game in CPN Tools as we will see later.

Every mortgage application passes through four stages:

1. initial processing: filling out the application with a loan officer
2. credit checking: confirmation of employment and review of credit history
3. surveying: a survey of the proposed property to check for its value
4. title checking: ensuring that the title to the property is uncontested

All the stages are modeled in an identical manner. Only the survey section of the model is described, to show how each stage's processing works. When an application is checked for the credit worthiness of its applicant, this application goes from the credit checking backlog into the surveying backlog. Every period, based on the backlog of surveys, the player sets the target capacity of the surveying stage. He can do so by deciding to hire or fire employees, in this case the surveyors. However it takes time to actually find, interview, hire or give notice to employees. The actual surveying capacity will lag the target surveying capacity by a fixed number of periods. This number has to be determined by the player himself, or an instructor. The surveyors that are currently employed will carry out as much surveys as possible for that period. In this game it is assumed that one surveyor completes one surveying task per period. Once the surveying task of an application is completed, this application will leave the surveying backlog and goes into the title check backlog. Each of the other three stage functions in the same way.

From the above we see that the only information available to the player that controls the surveying stage is the size of the backlog of surveys. Note that this is only the case when you play the game with a decentralized strategy.

For simplicity we assume that each application that enters the system is ultimately approved. This is off course not realistic; we would rather have some random survival rate for an application. In a real life situation this random survival rate complicates the management of a service supply chain. There is another factor that complicates the management of service supply chains. Each stage of the process is managed by a separate company. Each company controls its own capacity. A company only sees its own backlog when making the decision; it does not look at the application start rate or at other stage's backlogs. This creates something that looks like the bullwhip effect [3], with the difference that the inventory here is the capacity for a certain stage. Further it is not possible to stock goods in advance as a buffer for fluctuations in demand. Rather every stage must manage its backlog strictly by managing its capacity size. The capacity size is the number of workers it employs.

3.2. The CPN model of the Mortgage Game

3.2.1. Top level Petri Net

In figure 2 the top level Petri Net of the mortgage game is depicted. Stages 1,2,3 and 4 refer to the application processing, credit checking, surveying and title checking. Modeling this game in CPN was difficult. In [1] the game is described in detail. However it was not possible to model the game as described directly into CPN. The capacity adjustment and cost calculations in the CPN model differ from the original model.

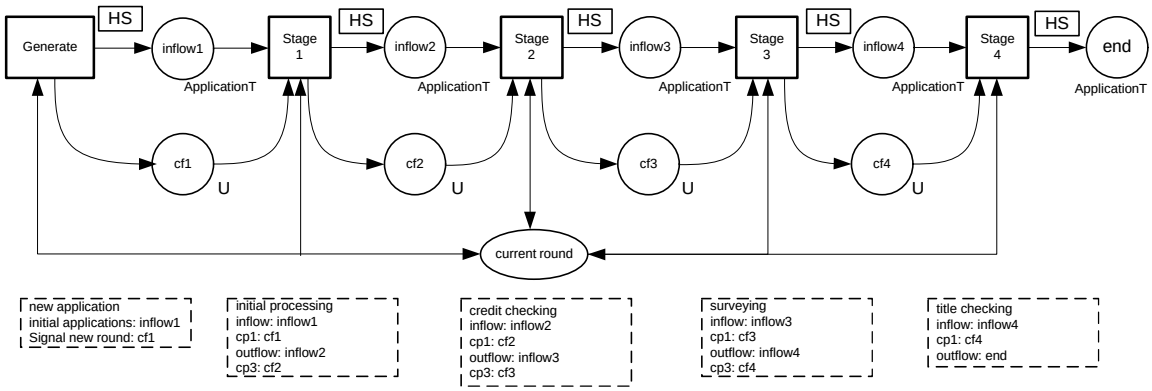

Figure 1 - Top level Petri Net of simplified mortgage game

3.2.2. Sub page generate

Transition *generate* generates new applications. The number of applications that arrive in one period is fixed for that period. If for instance on average 20 applications would arrive in one period, then generate puts a token with integer value 20 with a timestamp of that period in place *inflow1*. Another possibility for the generate transition would be to have applications arrive according to a negative exponential distribution. For this game however, this would be too complex, since it is not necessary to know when the 20 applications arrive. We assume that they arrive in one period, and we don't need to know the exact times. This is consistent with the assumptions of the authors of [1].

In the substitution transition the *new round* transition signals when a new round can start. In this model a round is equal to one time unit in CPN Tools. Initially there is one token of type Unit with timestamp 0 in place *init*. When the network connection is opened, transitions *generate* and *new round* are enabled. After firing, *new round* puts one token in place *signal new round*. In period t all transitions that are enabled in that period will fire. After this the period is t+1, and *new round* can fire again. The four stages can work on applications in parallel. They all process applications from their backlogs in the same period. To always have the same interleaving of stages a control token is passed through places *cf1*, *cf2*, *cf3* and *cf4*. When stage 4 fires, the control token in *cf4* is consumed, and *new round* can produce a new control token. These places are the control flow in the CPN model. Another possibility is to enable all stages at the same time t (instead of passing the token from stage to stage) and let the CPN Tools scheduler decide which transition can fire. When all transitions that are enabled at time t have fired, the time is increased to t+1 and a new round can start. For the game results it does not matter which the two possibilities are chosen, they are the same in both cases. However the latter possibility makes adding Comms/CPN functions difficult.

3.2.3. Sub page for the stages

Since all stages are modeled in the same way, we again only look at the surveying stage. The CPN model for the surveying stage is depicted in figure 3.

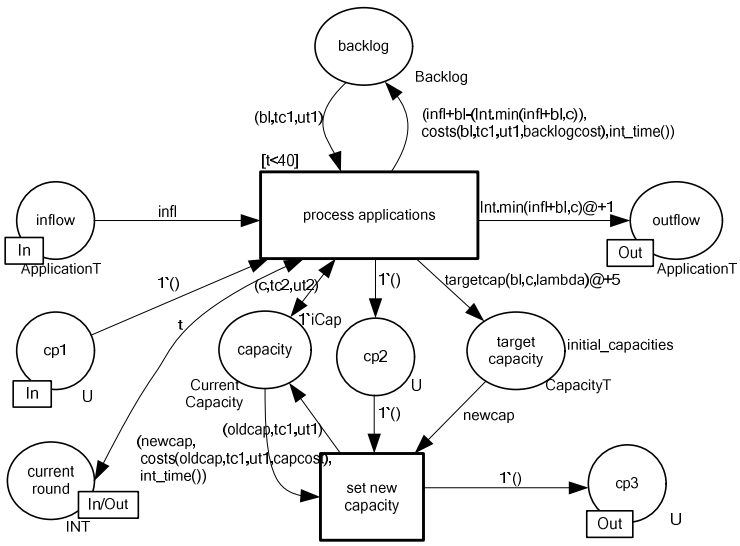

Figure 2 - Sub-page surveying

Transition *process applications* models the processing of the surveying task of an application. For simplicity assume that each employee has a productivity of one application per period. So the number of employees is the capacity. The completion rate of applications is equal to the minimum of the backlog plus any inflow from the previous stage or the capacity. For the most of the time the capacity will be the minimum.

The process *applications transition* is connected to three input/output places. Place *backlog* contains a token which holds the current backlog size, the backlog costs incurred until time t and the current time t. Place *current round* contains the current time of CPN Tools. Place *capacity* contains the capacity that is available for current round, the costs incurred until time t and the current time t. Transition *process applications* is enabled when there is a token in place *inflow* and a token in the control place *cp1*. The token in place *inflow* is the inflow for time t, the new applications that have arrived for the surveying stage. When transition *process*

applications fires it updates the backlog (number of applications and costs). Further a number of applications are processed and this number is put in place *outflow*.

Also *process applications* decides how much the preferred capacity had to be and produces a token having that value together with a timestamp t+5 attached to it. This token is placed in *target capacity*. *Process applications* also produces a token for *cp2*, enabling transition *set new capacity*. This place is unnecessary when the CPN model is used a simulation model. However when networking functionality is added it is necessary. When transition *set new capacity fires*, the capacity for period t+1 is set, and the costs of the capacity until period t are calculated. Also a token for place *cp3* is produced. The old capacity is consumed and the new capacity is set.

The target capacity for each stage will be set directly proportional to the stage's current backlog and inversely proportional to the service delay time λ. This is off course not an optimal policy, but it reflects reasonably well how real players make decisions in capacity management games. The λ represents how long it will take on average to complete a backlogged application.

As an example of how the capacity is determined, suppose λ is 10 periods. This means that one unit of capacity will need 10 periods to complete a backlogged application. Suppose the backlog is 200. The target capacity that will be available in 5 periods is equal to 200 divided by 10, which is 20. The target capacity is the average capacity per period necessary to process all backlogged applications within 10 periods. So this is a very simple rule, but according to [3] this is the best way for describing people's behavior in capacity management games.

Stages one through four are modeled as described above. Next the declarations for the *Mortgage Game* process model are explained. Then we will discuss how to turn the simulation model into a game by adding functions from the Comms/CPN library to the model.

3.3. Declarations in the CPN Model

In table 1 the declarations for the mortgage game process model are described.

Declaration	**Explanation**
color E = with E;	Standard declaration
color U = unit;	Unit type
color UT = U timed;	Timed Unit type
color INT = int;	Integer type
color BOOL = bool;	Boolean type
color STRING = string;	String type
color Application = INT;	Defines the application type
color ApplicationT = Application timed;	Timed application type
color Cost = INT;	Defines backorder and capacity cost type
color Capacity = INT;	Defines the capacity type
color CapacityT = Capacity timed;	Timed capacity type
color Period = INT;	Defines the period until which costs are calculated
color Backlog = product Application * Cost * Period;;	Defines the backlog type. The first element is the size of the backlog, the second element the backlog costs until time t, the third element is the current time t.
color CurrentCapacity = product Capacity * Cost * Period;	Defines the capacity type. The first element is the size of the capacity, the second

	element the capacity costs until time t, the third element is the current time t.
var bl: Application;	current backlog
var infl: Application;	number of applications that flow into a stage
var oldcap: Capacity;	old capacity
var newcap: Capacity;	the capacity that will be available in 5 periods
var c: Capacity;	current capacity
var t:INT;	current time of CPN model
var tc1, tc2: Cost;	tc1 and tc2 are used for the total cost until time t
var ut1, ut2: Period;	time until costs are calculated
var psc: Capacity;	The player set capacity
val iCap = (20,0,0);	The initial current capacity for each stage
val iBacklog = (200,0,0);	The initial backlog for each stage
val initial_inflow = 20;	The initial inflow for each stage
val capcost = 10;	Cost for 1 unit capacity per period
val backlogcost = 2;	Cost for 1 backlogged application per period
val lambda = 5;	Average completion time of an application, in this case this is 5 periods
fun int_time() = IntInf.toInt(time());	Typecasts the current time in CPN Tools from IntInf to Integer type
fun targetcap(bl,cap,lambda)= if int_time() mod 5 = 0 then round((Real.fromInt bl)/(Real.fromInt lambda)) else cap	Returns the capacity that is available in 5 periods. The new capacity is equal to the backlog size bl divided by the target lead-time lambda. To make the game more interesting, the nonplayer stages can only change their capacity every 5 periods. Cap is the current capacity.
fun costs(l,oc,t,cpu)=oc+ ((int_time()-t)*cpu*l);	Returns the new costs given an integer level l, the old costs oc, the until time t and the cost per unit cpu.
fun i_m(i:INT,j:INT,a:INT) = if i>j then empty else 1`(a)@+i ++ i_m(i+1,j,a);	Function i_m returns an initial marking
val initial_applications = i_m(0,4,20)++i_m(5,44,27);	This is the demand process of the mortgage game.
val initial_capacities = i_m(0,4,20);	The capacities that are available in the first 5 periods. The player's decisions only have effect form period 5 and later.

Table 1 - Declarations of the Mortgage game CPN

3.4. Turning the model into a game

In the *Mortgage Game* the player controls the credit checking stage. Substitution transition *generate* is changed. Before generate can start putting new applications into *initial applications*, the network connection is opened. CPN Tools will wait for an incoming connection request from an external program, in this case the Java application for the *Mortgage Game*. When the connection is established, *generate* can put new applications into *initial applications*.

When the connection is established, all the stages can send their status to the Java applet. Again look at figure 3. To send the status of a stage to the application a code segment is added to the *process applications* transition. The variables that are relevant for the status are in the

input part of the code segment. Functions provided by Comms/CPN handle the sending of the values of the variables over the connection. For the player controlled stage there is also an output variable in the code segment. This output variable is *psc* (player set capacity).The Java application sends a value for this variable to CPN Tools. Functions provided by Comms/CPN convert this value to the correct type. After the code segment has been executed, the variable has the value entered by the player in the application. This value is then put in place *target capacity* with timestamp t+5.

The *Mortgage Game* is modeled with a control token that is passed through places *cf1*, *cf2*, *cf3* and *cf4*. The control flow is grouped in the CPN Model. Stage 4 then consumes the control token so that *new round* can fire again. Since we always have the same interleaving of stages, the sending of the status of each stage can not happen one after the other. The reason for this is the receive command in stage 2. When the *process applications* transition of stage 2 fires, it sends the status of stage 2 to the application. It also blocks CPN Tools, because it causes CPN Tools to wait until it receives data from the application. After the value has been received from the Java application, stages 3 and 4 can send their status.

The game is played for 40 rounds. After 40 rounds the connection has to be closed in CPN Tools in order to be able to play the game again. Transition *new round* updates the time every round. The current time is put in place *current round*. Also connected to place current round is transition *close connection*. Transition guards are added to both transitions. *New round* can only produce new control tokens if the value of the token in place *current round* is smaller than 40. If the value of the token in place *current round* is equal to 40, *close connection* can fire. This ends the game by disabling the *generate* transition and closes the network connection.

Place *current round* is connected to each of the stages. A stage can check whether it still has to send/receive data from the Java application, by checking the current round number.

Initially the game starts in equilibrium. The first five periods the capacities and inflows are the same. This is because the decision made in the first round has its effect five periods later. This means that the place *target capacity* in figure 3 contains five tokens having a value for the new capacity. The generate transition generates 20 application starts per period for the first 7 periods. From period 8 this jumps to 27 starts per period, and stays at that level until the end of the game. This demand process is modeled by the tokens that are initially in place *get initial values*.

The target backlog delay λ is set to 5 periods. To make it easier for the player, the non player stages can only change their capacity every five periods. The player can do this every period. Further all the costs are set to 0. The time is also set to 0. Note that these values can be changed in the model.

3.5. Playing the game

The goal of the *Mortgage Game* is to minimize the total cost for the entire supply chain resulting from employee salaries and service delays. An employee costs 10 per period, whereas a backlogged application costs 2 per period. So hiring and firing costs aren't taken into account. This however doesn't have consequences for the game. In [1] these costs are taken into account. For simulation results of [1] refer to [4]. The player has to decide what capacity is going to be available in 5 periods. The current status of the player controlled stage is available to help in making the decision.

4. The Kanban Game

4.1. Description

The production system in this game is kanban controlled. This means that the number of products in between each production step is limited. It is only allowed to keep a certain number of units in stock, and a certain number of units to be produced. A kanban production system is an example of a pull controlled system[5][6].

[7] In the *Kanban Game* you control a work center that manufactures four products: A,B,C and D. To produce one of these products in the work center, the production line requires an elaborate setup every period. Changeover from product to another requires about the same required to produce one unit of product. Adding or removing employees from your work center changes capacity. Assume that the capacity can vary from 8 to 12 units per day. Startups and changeovers need to be subtracted from this capacity. Assume that both cost 1 unit of capacity.

As a consequence of the above a kanban system has been established. A small stock point is established near the work center. The management has calculated a certain maximum number of units allowed in the kanban stock point for each product.

Each period the customers come to the stock point and remove what they require for that periods shipment. For each product assume a demand process according to a discrete uniform distribution on the interval [1..3]. The average demand for each product is equal to 2 units per day. The withdrawal activity occurs while the production line is warming up and you must then decide what to produce during this period based on the kanban stock that remains.

In this game backorders are allowed. Backorders are represented by using negative kanban levels.

4.2. The CPN model of the Kanban Game

4.2.1. Top level Petri Net

In figure 4 the top level Petri Net of the *Kanban Game* is depicted. The customers are modeled in substitution transition *demand*. In this transition the customer requirements are generated and put in place *demand*. Further *demand* consumes the token in place *shipment*.

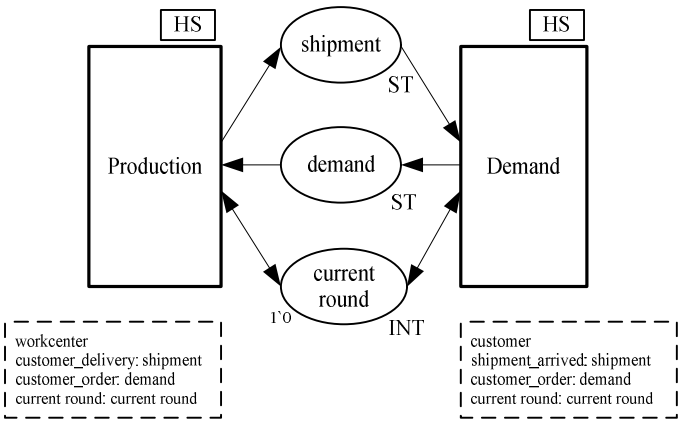

Figure 4 - Top page Petri Net of the Kanban game

Also a token that represents the current round is put into place *current round*. The current round is equal to the current time in CPN Tools.

The work center is modeled by substitution transition *production*. *Production* consumes from place *demand* the number of products that are required for that period and puts them in place *shipment*. In *production* the production mix for period t is determined. The production mix is a plan that contains how much to produce of each product A,B,C and D. Next the workcenter checks whether this production mix can be produced. This is done by checking if the stock levels after the production do not exceed the maximum level. Also it checks if there is sufficient capacity available to produce this mix. If the production mix can be produced, workcenter does so. The new products will be available on the start of the next period. When it is not possible to produce the production mix, production will again determine a production mix.

4.2.2. Sub page consumer

Figure 5 depicts the sub page consumer. The dotted transitions and places are necessary for the communication between CPN Tools and the Java application. This will be discussed later. Transition *consume* accepts the customer delivery of place delivery. Transition *generate demand* generates the customer demand for each period. Initially there is 1 token in place *init* and one token in *current round* with integer value 0. When *generate demand* fires it consumes the token in *current round*. It produces one token for place *customer order*. This token contains the demand for products A,B,C and D for period t. The current time of CPN Tools is put into place *current round*. The customer order is delivered in the same period as it was ordered. This is a reasonable assumption, because delivery times are not an essential part of the *Kanban Game*. Because of this assumption and new round being the only timed place in the model, the clock is increased when transition consume fires.

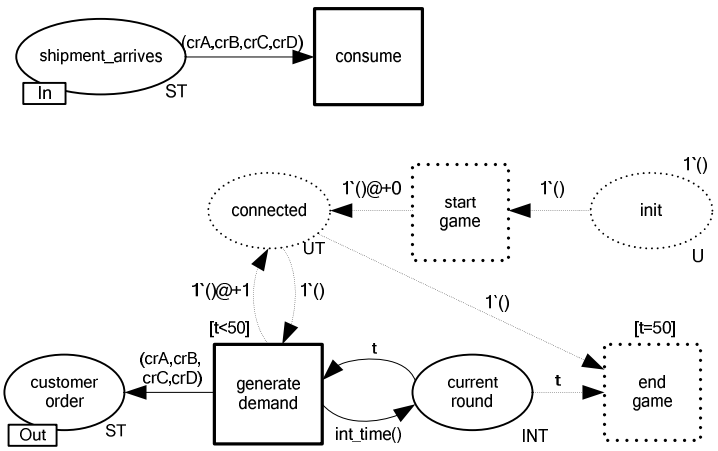

Figure 5 - Sub page consumer

4.2.3. Sub page workcenter

In figure 6 we see the sub page for the *workcenter* substitution transition. The two places *stock* are in a fusion set [8] and can be seen as one place representing the kanban stock positions for the four products. When there is a token in place *customer order*, *ship goods* can fire and the whole order is delivered. Ship goods also produces a token for place signal start. This also happens when there is not enough kanban stock available. However the kanban

stock for that product will become negative. This assumption works for the kanban game, because a negative kanban position indicates that a customer order was not fulfilled (i.e. backordered). It needs to be fulfilled as soon as possible. Transition *determine production mix* produces a token for place *prodmix*. This token contains the production mix for the current round. At this point two things can happen; the production mix is ok or is not ok.

This can be modeled by one transition with a code segment attached to it. In this code segment the production mix is checked and the correct output is produced. However by modeling it this way, it becomes very difficult to add Comms/CPN functions to the model. Also the programming of the application belonging for this game becomes very hard, because the communication between the application and CPN Tools becomes complex.

Therefore the checking of a production mix was modeled by two transitions: *mix ok* and *mix not ok*. By adding transition guards, only one of them will be enabled when there is a token in place *prodmix*. The guard of transition *mix not ok* is the negation of the guard of *mix ok*. The guard consists of two checks:

- Does the production mix match the available capacity ?
- Is the maximum kanban stock of each product not exceeded ?

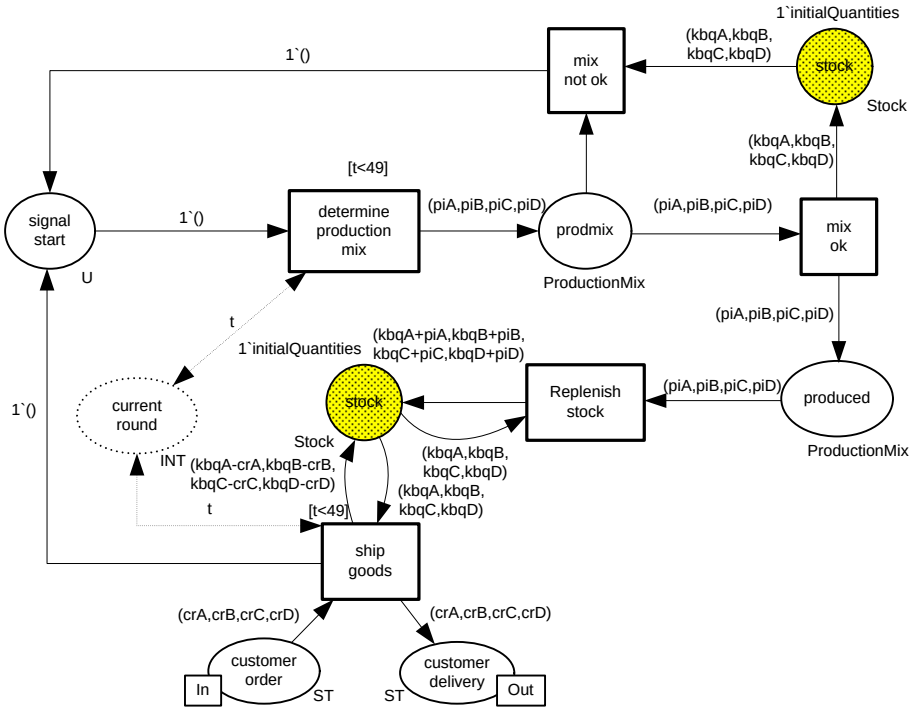

Figure 6 - Sub Page work center

Therefore the kanban stock has to be connected to both transitions. When a production mix is not ok, it has to be determined again. When the production mix is ok, it is produced. This is modeled by place produced. Now transition *replenish stock* is enabled and the new stock positions are calculated.

4.3. Declarations in the CPN model

Next a list with declarations in CPN Tools is given together with a description for each declaration. Then we discuss how to transform the model into a game.

Declaration	**Explanation**

color E = with e;	Standard declaration
color U = unit;	Unit type
color UT = U timed;	Timed Unit type
color INT = int;	Integer type
color INTT = INT timed;	Timed Integer type
color BOOL = bool;	Boolean type
color STRING = string;	String type
color Quantity = INT;	Defines the quantity type of a product
color Stock = product Quantity * Quantity * Quantity * Quantity;	Defines the stock type, each element of the product represents a kanban stock level.
color Sales = product Quantity * Quantity * Quantity * Quantity;	Defines the Sales type, each element of the product represent how much of each of the products is required
color ST = Sales timed;	Timed Sales type
color ProductionMix = product Quantity * Quantity * Quantity * Quantity;	Defines the production mix, each element of the product represents how much to produce of each product.
val maxKanbanStock = 8;	constant, maximum stock for each product
val maxCapacity = 12;	constant, maximum available capacity
val initialQuantities = (6,6,6,6);	The initial kanban stock for each product
var kbqA,kbqB,kbqC,kbqD:Quantity;	the kanban stock quantities for each of the products
var piA,piB,piC,piD:Quantity;	units produced of each of the products
var crA,crB,crC,crD:Quantity;	requirements for each product
var t: INT;	time of CPN model
fun changeOver(x:INT) = if x>0 then x+1 else x	Returns the capacity needed to produce a certain number of units of one product. Auxiliary function of totalCap
fun totalCap(q1,q2,q3,q4) = changeOver(q1) + changeOver(q2) + changeOver(q3) + changeOver(q4);	Returns the total capacity required to produce a certain production mix.
fun maxStock(x:INT,y:INT) = if x<=y then true else false	Returns whether the maximum kanban stock for one product is exceeded or not. Auxiliary function for exceedMaxStock.
fun exceedMaxStock(s1,s2,s3,s4) = maxStock(s1,maxKanbanStock) andalso maxStock(s2,maxKanbanStock) andalso maxStock(s3,maxKanbanStock) andalso maxStock(s4,maxKanbanStock);	Returns whether all stock positions are ok. Used in the transition guards of mix ok and mix not ok.
fun capOK(x1,x2,x3,x4) = if totalCap(x1,x2,x3,x4) <= maxCapacity then true else false	Returns whether the maximum available capacity is exceeded or not. Used in the transition guards of mix ok and mix not ok.
fun newDemand() = (discrete(1,3), discrete(1,3), discrete(1,3), discrete(1,3));	Returns the demand for the current period.
fun int_time() = IntInf.toInt(time());	Typecasts the current time in CPN Tools from IntInf to Integer type.

Table 2 - Declarations of the Kanban game CPN

4.4. Turning the model into a game

In the *Kanban Game* the player has to decide how many units of each of the products have to be produced in a period. Again look at the sub page *consumer* depicted in figure 5. Before transition *generate demand* can fire a network connection needs to be opened. Initially there is a token in place *init*. At time 0 only transition *open connection* is enabled. When *open connection* fires, the network connection is opened. CPN Tools will wait for an incoming connection request from an external program, in this case the Java application for the Kanban Game. When the connection is established, *open connection* produces a token for place *connected*. Also the current status of the game is sent to the application. Now transition *generate demand* is enabled. Every round *generate demand* sends the current model time and the demand for each of the product to the Java application.

In the sub page workcenter, the transition *ship goods* sends the current kanban stock position to the Java application. This enables transition *determine production mix*. Now CPN Tools is blocked until it receives a production mix from the application.

When the production is incorrect, transition *mix not ok* will send integer value 0 to the application to indicate the production mix is incorrect. It also sends a message that the application will display to inform that the production mix is incorrect. Next a new production mix needs to be determined.

If the production mix is correct, transition *mix ok* will send integer value 1 to the application to indicate that the production mix is correct. Also a message that the application can display is sent.

The *Kanban Game* is played for 50 rounds. After 50 rounds the game ends and the network connection has to be closed. This is modeled in the *customer* substitution transition. When transition *end game* fires, the token that is in place *connected* is removed. This disables transition *generate demand*, and the connection is closed. This is modeled with transition guards. *Generate demand* can only fire when current round is smaller than 50, while end game can only fire when the current round is equal to 50. Because the game is modeled with two substitution transitions, *workcenter* can still fire when the connection is closed. To solve this, place *current round* was parameterized. Substitution transition *demand* updates place *current round*, while *workcenter* uses this place to check in which round the current game is. Also a transition guard was added to transition *determine production mix*, enabling it only when current round is smaller than 50. This construction ends the game in a proper way.

4.4. Playing the game

The game is played as follows. Initially the stock level for all products is set to 6. The day's requirements as well as the resulting stock level appear in the application. Now the player has to determine a production mix, taking into account the available capacity and the maximum kanban stock. As stated before a production mix has to be such that both constraints are not exceeded. The reasons for this are:

- Overproduction implies overtime. Often this is acceptable, but in this it is not allowed. If there is overproduction either the number of changeovers has to be lowered and/or the number of production units.
- Overstocking is not allowed. When overstocking happens, the production mix for that day has to be changed.
- Negative kanban stock positions indicate that delivery promises have not been met. For simplicity backorders are allowed in this game. The number of back orders is a useful performance measure.

While playing a graph appears displaying the total stock level and the stock level for each of the products A,B,C and D. These graphs need to be used for the analysis. The game has to be played four times using the scenarios listed below.

Scenario	Capacity	Stock
1	12	12
2	10	12
3	12	8
4	10	8

Table 3 - Different scenarios

These scenarios need to be entered into the CPN model of the *Kanban Game*. The two constants maxKanbanStock and maxCapacity need to have the correct values.

After the game has been played, try to answer the following questions for each of the scenarios:

- What effect does kanban stock level have on system performance?
- What effect does capacity have on system performance?
- How do capacity and kanban stock interact?
- Are there alternate daily scheduling strategies that we might have used to improve performance?
- How do daily scheduling strategies relate to capacity and maximum stock level?

5. The Distribution Game

5.1. Description

The distribution game [9] is about the problems of ordering and allocating stock in multi-level distribution systems. There are two stocking levels in this distribution system: the central warehouse and the retailers. The player controls both levels and has to make the following decisions:

- When to order and how much to order from the supplier.
- When to ship and how much to ship to each retailer.

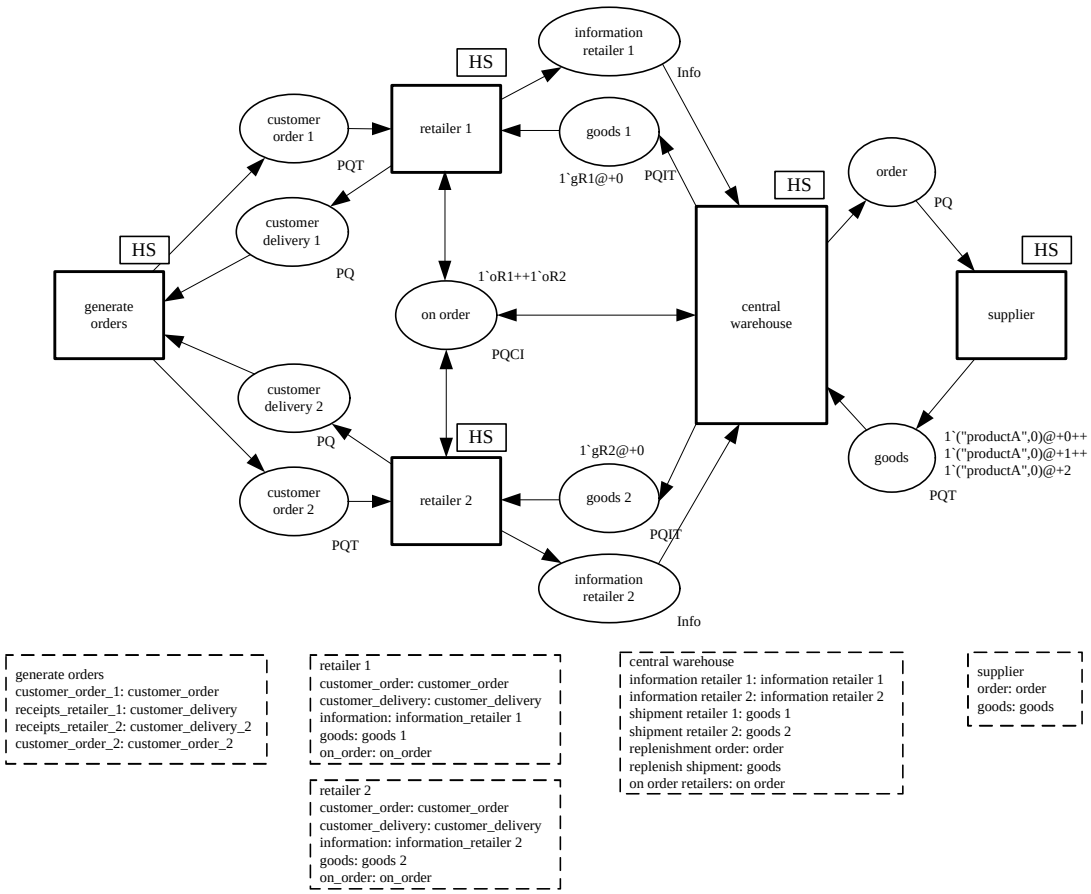

Figure 7 - Top level Petri Net of distribution game CPN model

Customers buy the stock from the retailers and the objective is to make as much money as possible from these sales. You cannot sell something you do not have. So make sure to keep the retailers supplied with stock. Do not place orders or make shipments too frequently or your order costs will be too high. But keeping too much inventory in the system will make holding costs too high. Every review period of the game you have to make three decisions:

- how much to order from the supplier
- how much to ship to retailer 1
- how much to ship to retailer 2

When making these decisions look at the order and holding costs. Also it takes 3 three periods before the goods ordered from the supplier arrive at the central warehouse, and it takes one period to ship goods to each of the retailers. Further assume that the information lead-time is this game is zero.

The assumptions for this game are that supplier has unlimited manufacturing capacity, and that there are no backorders allowed. The latter means that the retailers have complete lost of sales in case of a stock out.

5.2. The CPN model of the Distribution Game

5.2.1. Top level Petri Net

In figure 7 the top level Petri Net of the distribution game is depicted. Substitution transition *generate orders* generates the customer demand. The demand process for both retailers is a discrete uniform distribution on the interval [3..17] for retailer 1; and on the interval [1..15] for retailer 2. The demand for period t is put in the places *customer order 1* and *customer order 2*. The *supplier* substitution transition consumes from place *order* an order quantity for a certain product type. This token is passed to place *goods* with delay 3. This models the lead-time from the supplier to the central warehouse.

Place *on order* is connected to both of the retailers and the central warehouse. Central warehouse places the shipment quantities for both of the retailers in place *on order*. Now the retailers know how much they still have to receive. When a retailer receives its shipment it decreases the value of *on order*. *On order* is decreased with the quantity of the received goods (places *goods 1* and *goods 2*). As stated earlier, each retailer keeps its own stock. Every review period the retailers send information about their current status to the central warehouse. This information can help in making the ordering decisions.

5.2.2. Sub page retailers

In figure 8 the sub page for retailer 1 is depicted. The sub page for retailer 2 is the same. Assume that before the retailer checks the stock to fulfill a customer order. However the stock needs to be replenished first. Now the retailer has the most recent stock position available to determine the quantity shipped to the customer. This order of activities is modeled with places *mutex1* and *mutex2*. Initially the control token is in place *mutex1*, enabling the *replenish* transition. When transition *replenish* has fired, the control token is put in place *mutex2*. This enables *deliver order*.

Transition *replenish* handles the incoming goods. When *replenish* fires, the value of the token in place *on order* is decreased with the number of goods received. The goods are added to the *stock* and the holding costs of the stock until period t are calculated. Also a control token for place mutext2 is produced, indicating that an incoming customer order can be handled.

The *deliver order* transition determines what quantity of goods is delivered to the customer. The strategy used is to deliver the whole customer order if possible. When this is not possible (i.e. the stock is not sufficient), the size of the customer delivery is equal to the number of units in stock. The stock position then becomes equal to zero. This means that there are no backorders allowed.

Also the costs of goods sold and the revenue are calculated when transition *deliver order* fires. The costs and revenue until time t are in place *costs and revenue*. When *deliver order* fires, these values are updated.

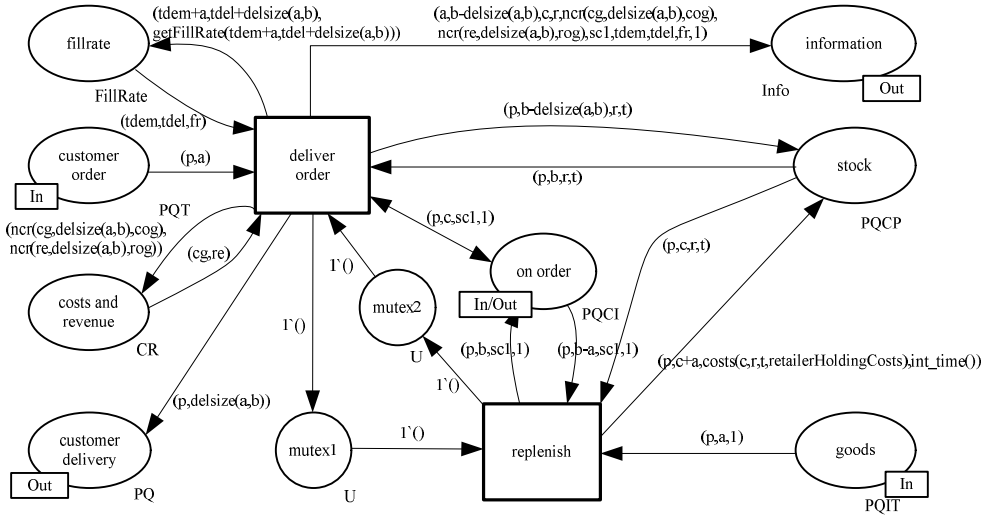

Figure 8 - Sub page retailer 1

An interesting performance measure in a distribution chain is the *end customer fill rate* [17]. It is the percentage of customer demand which is satisfied immediately off-shelf. Logistics is generally a cost-center service activity[9], but it provides value via improved customer satisfaction. It can quickly lose that value if the customer becomes dissatisfied. The end customer can include another process or work center inside of the manufacturing facility, a warehouse where items are stocked or the final customer who will use the product.

Besides calculating and updating the status of the retailer, *deliver order* also sends the current status to the central warehouse. The retailer status consists of: *current customer demand, stock position, on order position, incurred holding costs, total cost of goods sold, total revenue, shipment costs, total demand, total delivered, fill rate* and *its identity*. A token containing these values is put in place *information*. The identities of retailers are modeled with integer numbers.

5.2.3. Sub page central warehouse

Figure 9 depicts the warehouse sub page. The warehouse has two functions:

- collecting information about the retailers and replenishing the warehouse stock
- determining shipment sizes to the retailer and placing orders with the supplier

These two function are modeled with substitution transitions collect information and determine order sizes. In figure 9 the sub page for the central warehouse is depicted. *Collect information* handles shipments from the supplier and sends the status of the central warehouse and both retailers to the Java application. Determine order sizes receives from the Java application the shipment and order sizes. Place *cp2* is added because *determine order sizes* is only allowed to fire after *collect information*. If this place is not added, then *collect information* can report the wrong status to the Java application.

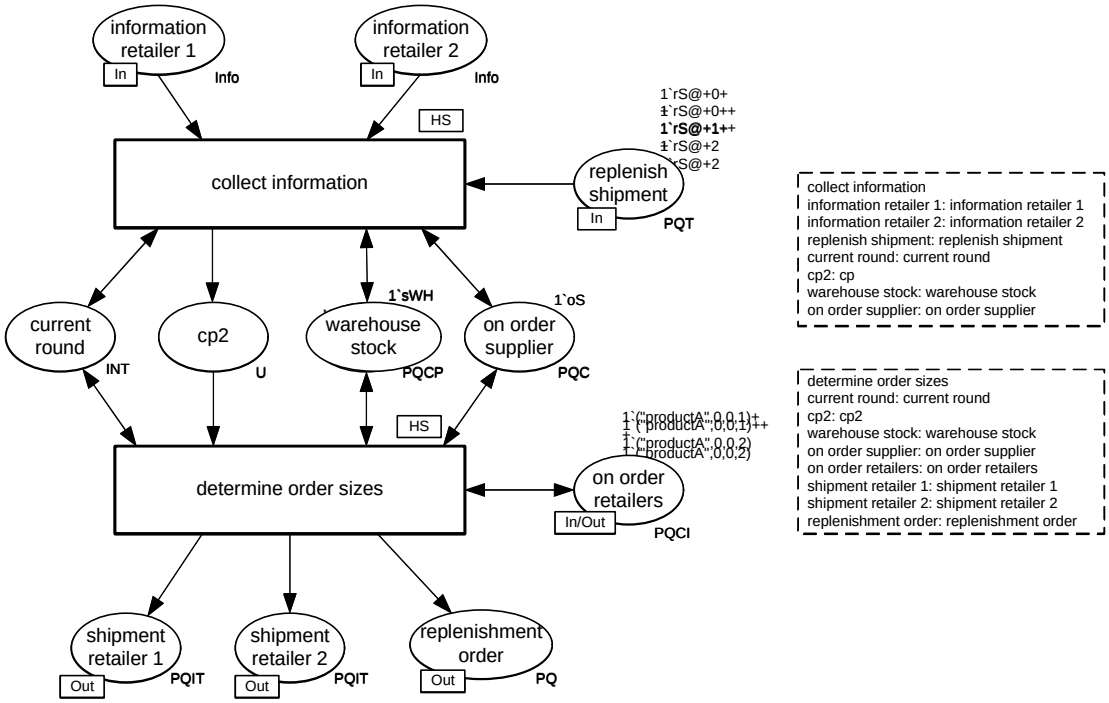

Figure 9 - Sub page central warehouse

Next both sub pages are discussed.

5.2.3.1. Sub page collect information

The sub page *collect information* is depicted in figure 9a. Assume that the stock is replenished before the status is reported. This is modeled the by adding place *ct1* between transitions *replenish* and *report status*. After the stock has been replenished transition *collect info* is enabled. The statuses of the retailers and the central warehouse are sent to the Java application. Also a token for place *cp2* is produced.

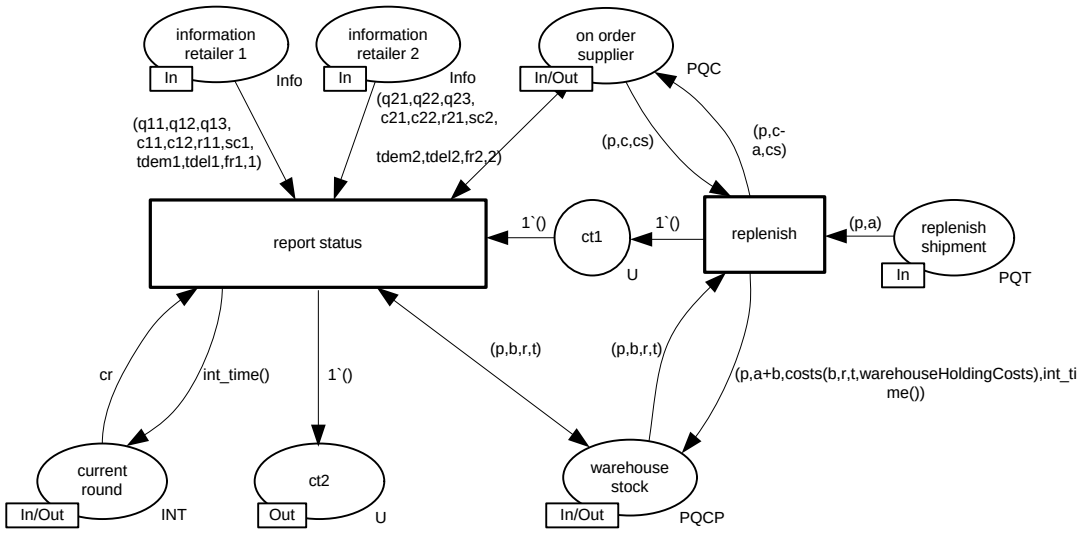

Figure 9a - Sub page collect information

This enables *determine order sizes*.

5.2.3.2. Sub page determine order sizes

Figure 9b depicts the sub page *determine order sizes*. When *get order sizes* fires, CPN Tools is blocked until it receives the shipment and order sizes from the Java application. The shipment and order sizes are put in place *order sizes*.

Next the order sizes have to be checked. There are two types of order sizes:

- the size of the order placed with the supplier
- the sizes of the shipments to the retailers

Orders for the supplier are always correct because the supplier has unlimited manufacturing capacity.

The sum of the retailer shipments cannot exceed the warehouse stock, it is impossible to deliver more than there is in stock. This is modeled by adding transition guards that exclude each other to transitions *order sizes nok* and *order sizes ok*. This construction was also used with the *Kanban Game*. When the shipment sizes are incorrect *order sizes nok* will fire. The shipment and order sizes have to be determined again. When the order sizes are correct *order sizes ok* will fire. The number of goods on order for each retailer are updated, along with the shipment costs. Also the number of goods on order with the supplier are updated, along with the supplier order cost. Further the order is placed with the supplier, and the goods are shipped to the retailers. It takes 1 period to ship goods from the central warehouse to the retailers. This is modeled by adding a delay of 1 to the tokens that are produced for places goods ret 1 and goods ret 2.

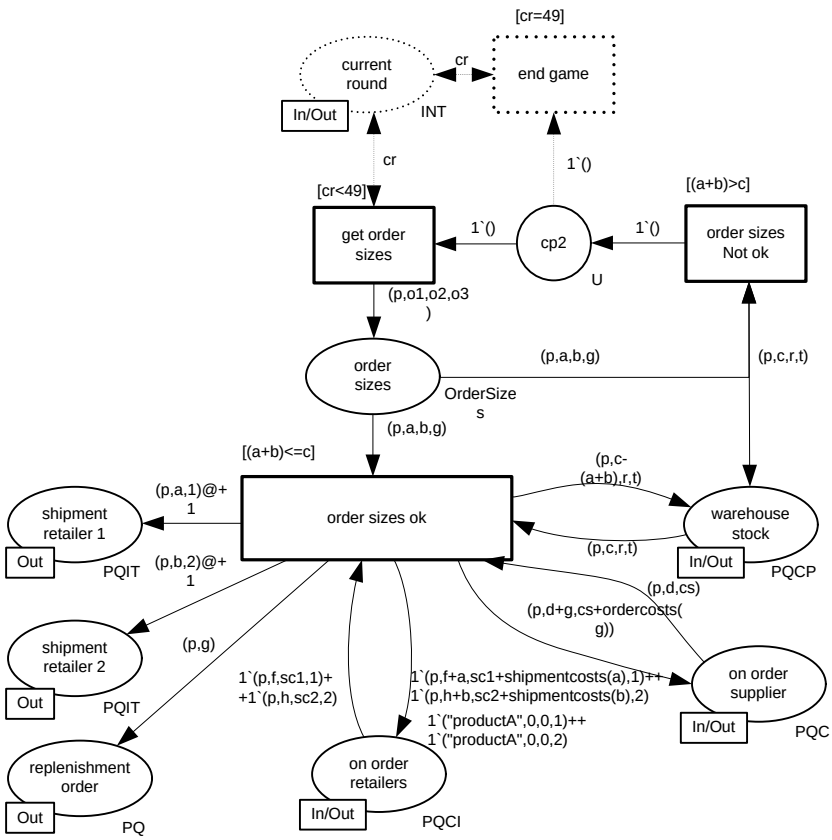

Figure 9b - Sub page determine order sizes

5.3 Declarations in the CPN model

Next a list with declarations in CPN Tools is given together with a description for each declaration. Some explanations are general, because variables and functions are used more than once.

Declaration	**Explanation**
color E = with e;	Standard declaration
color U = unit;	Unit type
color UT = U timed;	Timed Unit type
color INT = int;	Integer type
color BOOL = bool;	Boolean type
color STRING = string;	String type
color Product = STRING;	Defines the product type, different products have different names.
color Quantity = INT;	Defines the quantity
color Cost = INT;	Defines the costs. Costs can be holding costs, shipment costs and order costs.
color Period = INT;	Defines the period type
color RID = INT;	Defines the retailer identity type.
color PQCP = product Product * Quantity * Cost * Period;	Defines the type used for the stocks.
color PQ = product Product * Quantity;	Used for the orders from the central warehouse to the supplier
color PQT = PQ timed;	Used for the shipment from the supplier to the central warehouse and for the customer demand.
color Revenue = INT;	Defines the revenue type
color CR = product Cost * Revenue;	Used for the cost and revenue of goods sold calculations
color OrderSizes = product Product * Quantity * Quantity * Quantity;	Product is the identifier, the three quantities represent the shipments to the retailers and the supplier order.
color PQI = product Product * Quantity * RID;	A shipment is determined by its product and retailer id.
color PQIT = PQI timed;	Timed PQI type
color PQCI = product Product * Quantity * Cost * RID;	Used for the on order place. The shipment costs per retailer per product can be recorded.
color TotalDemand = INT;	Defines the total demand
color TotalDelivered = INT;	Defines the delivered quantity
color Percentage = INT;	Defines the percentage, used to represent the fill rate
color FillRate = product TotalDemand * TotalDelivered * Percentage;	The fill rate can be calculated using total demand and total delivered. The calculated fill rate is represented using the percentage type.
color Info = product Quantity * Quantity * Quantity * Cost * Cost * Revenue * INT * TotalDemand * TotalDelivered * Percentage * RID;	This is the most important color, it defines which information of the retailers is to be send to the central warehouse.
val sWH = ("productA",100,0,0);	The initial warehouse stock
val sR = ("productA",20,0,0);	The initial retailer stock
val rS = ("productA",0);	Initial shipment to from supplier to the

	central warehouse
val oS = ("productA",0,0);	Initial on order with supplier
val oR1 = ("productA",0,0,1);	Initial on order for retailer 1
val oR2 = ("productA",0,0,2);	Initial on order for retailer 2
val gR1 = ("productA",0,1);	Initial shipment for retailer 1
val gR2 = ("productA",0,2);	Initial shipment for retailer 2
val cog = 70;	Constant, costs of one unit of product
val rog = 100;	Constant, revenue of one unit of product
val retailerHoldingCosts = 2;	Holding costs for unit per period at the retailer level
val warehouseHoldingCosts = 3;	Holding costs for unit per period at the central warehouse level
var p:Product;	Variable used to get the correct product.
var a,b,c,d,f,g,h:Quantity;	Variables used in calculations of quantities
var r,cg:Cost;	Variable r represents costs incurred until time t, variable cg is used in cost of goods sold calculations.
var t:Period;	The period until which the costs are calculated.
var re:Revenue;	re is used in revenue calculations
var cs,sc1,sc2:INT;	cs are the order costs, sc1 and sc2 the shipment costs to the retailers.
var q11,q12,q13,q21,q22,q23,o1,o2,o3:Quantity;	o1,o2,o3 store the quantities received from the Java application. The variables starting with q store information about the retailers.
var c11,c12,c21,c22:Cost;	These variables represent the costs incurred by the retailers.
var r11,r21:Revenue;	r11 contains the revenue of retailer 1 r21 that of retailer 2
var cr:INT;	Time of CPN model / The current round
var tdem,tdem1,tdem2:TotalDemand;	tdem is used in fill rate calculations; tdem1 and tdem2 are the total demand for each of the retailers.
var tdel,tdel1,tdel2:TotalDelivered;	tdel is used in fill rate calculations, tdel1 and tdel2 are the total delivered quantity for each of the retailers.
var fr,fr1,fr2:Percentage;	fr is used in fill rate calculations; fr1 and fr2 are the current fill rates of the retailers.
fun delsize(q,s) = if q>s then s else q	Returns the size of the customer delivery. Deliver the whole order. When the stock is not sufficient, deliver all that is on stock.
fun ncr(m,n,c) = m + n *c;	Returns the new costs/revenue of goods sold.
fun int_time() = IntInf.toInt(time());	Typecasts the current time in CPN Tools from IntInf to Integer type.
fun costs(l,oc,p,cpu)= oc+((int_time()-p)*cpu*l);	Returns the costs incurred until time p.
fun ordercosts(x) = if x =0 then 0 else 25	Returns the supplier order costs. If there is nothing ordered, i.e. order size o3 is 0, then the costs have to be 0.
fun shipmentcosts(x) = if x=0 then 0 else 3	Returns the retailer shipment costs. If there is nothing shipped, i.e. shipment size o1/o2 is 0, then the costs have to be 0.
fun getFillRate(demand,delivered)=	Returns the fill rate. Fill rate is an integer

floor(((Real.fromInt delivered)/(Real.fromInt demand))*100.0);	percentage between 0 and 100. The fill rate is calculated using the total demand and the total delivered number of goods.

Table 4 - Declarations of the distribution game CPN

5.4 Turning the model into a game

In the *Distribution Game* the player has to decide how much to ship to each of the retailers and how much to order from the supplier. Before *generate* can start generating demand, the network connection has to be opened. This is done in substitution transition *generate*. CPN Tools is blocked until it receives a connection request from the Java application. When the application has connected to CPN Tools, *generate* can start generating demand.

The *Distribution Game* is played for 50 rounds. Transition *report status* sends the statuses of the retailers and the warehouse to the application. Further it also updates the current round. Transition guards that exclude each other are added to transitions *end game* and *get order sizes*.

To model the ending of a game, the dotted part in figure 9b was added. When the value of the token in *current round* is equal to 49, transition end game is enabled. When *end game* fires, the control token in place *cp2* is removed and the network connection is closed. A guard also needs to be added to the *generate* transition, disabling it when the end of the game is reached.

All the interaction with the Java application is done in sub page central warehouse. The central warehouse receives from the retailers their statuses and sends them together with its own status to the Java application.

After the statuses have been sent, transition *get order sizes* will fire. CPN Tools can now receive the shipment and order sizes from the application. Now there are two possibilities, the sizes can be correct or incorrect. To indicate to the Java application that the order sizes are incorrect, transition order sizes nok sends integer value 1, followed by a message which says that the order sizes are incorrect. The Java application then displays this message. The same happens when the order sizes are correct. The difference is that in this case integer value 0 is sent to the application together with a message which says that the order sizes are correct. When the order sizes are correct the game advances to the next round.

5.5. Playing the game

In order to play the game, the model needs to contain a couple of tokens initially. The stock position of both of the retailers is set to 20, whereas the stock position of the central warehouse is set to 100. All costs, revenues and counters are set to 0. The fill rates are set to 100 percent. There is one token in place *goods 1*, with value 0 and delay 0. Also there is a token in place *goods 2*, also with value 0 and delay 0. Because of this the value for on order is also 0. Further there have to be three tokens in place *goods*, all having value 0, with delay 0,1 and 2. The reason for this is that the first decision to place an order with the supplier is made in period 0. Because there is a lead-time of 3 periods, the earliest time that a token is available in place goods would be 3. The table 5 lists the costs and lead-times together with their values.

Cost/Lead-time	Value
Cost of goods sold	70
Revenue of goods sold	100
Retailer holding cost	2
Central warehouse holding cost	3
Order cost	25
Shipment cost	3
Lead-time from warehouse to retailer	1
Lead-time from supplier to warehouse	3

Table 5 - Cost and lead-time values

These values can be changed in order to change the game. This game can also be used to check what happens with the costs when information is globally available or not. By changing the model, the player can determine which information is available to him. Also the retailer sub pages can be changed to incorporate backorders, instead of only partial shipments.

The goal of the game is to make as much profit as possible.

6. The Beer Game

6.1. Description

The *Beer Game* [10] is a role-playing simulation developed at MIT in the 1960's to clarify the advantages of taking an integrated approach to the managing of a supply chain. Consider a simplified beer supply chain. This supply chain consists of a retailer, a wholesaler which supplies the retailer, a distributor which supplies the wholesaler and a factory with unlimited capacity and raw material which brews the beer and supplies the distributor. Each component in the supply chain has unlimited storage capacity and there is a fixed supply lead time and order delay time between each component. This supply chain is depicted in figure 10.

Each week, every component in the supply chain tries to meet the demand of the downstream component. Any orders which cannot be met or fulfilled are recorded as backorders, and are met as soon as possible. No orders will be ignored and all orders must eventually be met.

At each period every component in the supply chain is charged a shortage cost of 2 per backordered item. Also, every period each location is charged inventory holding cost 1 per inventory item it owns. Each component in the supply chain owns the inventory at that facility. Assume that the items that are in transit between components have cost zero.

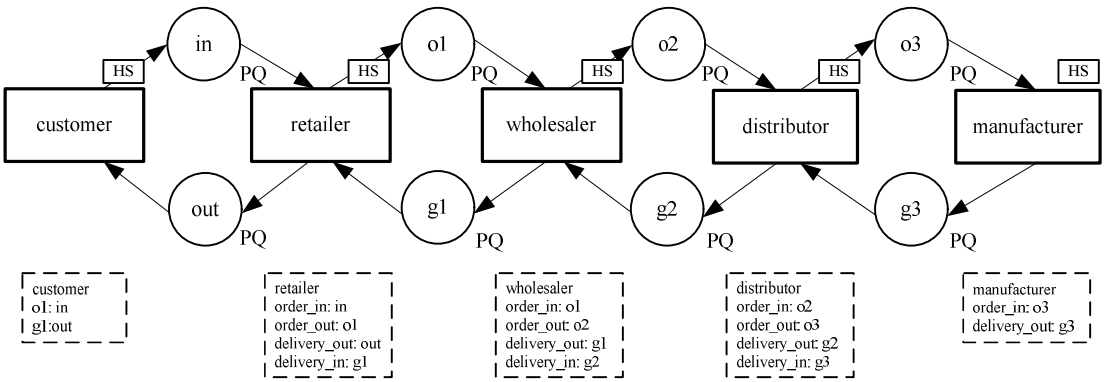

Figure 10 - Top level Petri Net of the beer game model

Each supply chain member orders some amount from its upstream supplier. It takes one week for this order to arrive at the supplier. Once the order arrives, the supplier attempts to fill it with available inventory, and there is an additional two week transportation delay before the material being shipped by the supplier arrives at the customer who placed the order. In figure 10 the orders that one component can place with the component upstream are modeled with places *o1*, *o2* and *o3*. Because there is an order delay time, these places have to be timed. This is also the case for places *g1*, *g2* and *g3*. These model the goods in transit, and are also timed places. Customer demand and deliveries are modeled by *customer*.

The goal of the retailer, wholesaler, distributor and factory is to minimize cost. They can choose to do this individually or for the entire supply chain.

6.1.1. The Bullwhip effect

Important processes in a supply chain are the ordering and delivering of goods. If these processes are disordered, this can lead to unwanted effects in the supply chain[11]. An example of a disorder is a sudden in change the end customer demand. This can happen because of [12,p472]: trade promotions, volume discounts, long lead times, full truckload discounts and end of quarter sales incentives. The consequence is that the orders seen at the manufacturer are highly variable. In fact the variability in order sizes increases in moving up the supply chain from customer to retailer, from retailer to wholesaler, from wholesaler to distributor and from distributor to manufacturer. This is called the Bullwhip Effect. Orders seen at the higher levels of the supply chain exhibit more variability than those at levels closer to the customer. The Bullwhip Effect can be simulated using the CPN model of figure 9. The results are in appendix A4. According to [12,p473-474] there are four rational factors that help to create the Bullwhip Effect:

- Demand signal processing: if demand increases, firms order more in anticipation of further increases, which results in an artificially high level of demand.
- Rationing game: there is a shortage, so a firm orders more than the actual forecast, hoping that it receives a larger share of the items that are short in supply.
- Order batching: fixed costs at one location lead to batching of orders.
- Manufacturer price variations: these encourage bulk orders.

The last two factors generate large orders followed by small orders. This implies increased variability at upstream locations.

In order to simulate the *Beer Game*, the components in the supply chain have to make a decision of how much to order upstream. The idea is to have to components make order decisions according to a smart rule. But since the components are linked, it turns out that it wasn't a smart rule after all. The Bullwhip Effect occurred.

6.2. The CPN model of the Beer Game

6.2.1. Order strategies

The strategy to keep the stock at a certain level is almost the ideal strategy. This strategy can be realized by using an (s,S) system. Every time an order decision has to be made, look at the current stock position, if it is lower than s, place an order that will fill the stock to point S. If it is above s, then do nothing. How to model this in CPN can be found in [5,p217]. Because of the assumption of unlimited storage capacity, this strategy was not used in the Beer Game CPN model.

Another strategy [11] is to pass the order 1:1 from component to component. If the customer order can be fulfilled from stock, then an order equal to that customer order will be placed at the upstream component. If the stock is not sufficient, then an order equal to customer order minus current stock plus two times that customer order (to cover the lead time of two periods) will be placed at the upstream component. The problem with this strategy is that the stock at the components never shrinks, it keeps growing. This is not a realistic situation. Also the Bullwhip Effect will clearly happen with this strategy.

The above strategies do not take into account demand history and do not have a service level. Therefore a strategy is designed that does take this into account. The sub page modeling the

components of the supply chain is depicted in figure 11. In explaining the strategy there will be referred to this sub page. The strategy compares the actual economic inventory level of a component with a preferred inventory level. If the actual level is below the preferred level, then an order of size (*preferred level - actual level*) is placed at the component upstream.

The actual economic inventory for period t is equal to (*stock + on order – pending orders*). This value is the same regardless of the order in which the three transitions (*handle_orders*, *handle_delivery* and *deliver*) fire in period t.

The preferred inventory [6,p24] level for period t is determined by:

*ceil(delay * moving average + security factor * standard deviation)*

- The delay is the fixed supply lead-time and is set to 3.
- The security factor determines the service level. is set to 1.645, which means that the components have a 95% service level.
- Standard deviation is the standard deviation that belongs to moving average [6,p82]. Assume that the standard deviation is calculated over the demand history.

The calculation of the moving average is programmed in CPN ML. Since reals are not allowed in CPN Tools, all the necessary calculations are done in CPN ML and the final value is converted to integer. The functions for calculating the moving average and the standard deviation take as input a list of integers. This list represents the demand history, and is stored in place history. The player can determine how many periods history there are. The more periods there are, the better the forecast will be. In appendix A4 the simulation was run with a moving average based on 4 periods with a 95% service level. If the orders are at the same level for a long time interval, the standard deviation becomes smaller, also the safety stock becomes lower. When there are variations in the order sizes, the safety stock becomes higher.

6.2.2. Sub page component

When the demand for period t is in place *order_in*, transition *handle_orders* can fire. It determines the order to be placed at the next component upstream. It does so by using the strategy described above. The order is then put in place *order_out*, and also the *total on order* position is updated. The demand is then added to place *pending orders*. The transition guard of transition *deliver* makes sure this transition is only enabled when there are *pending orders* and when the *stock* is greater than zero. When *deliver* fires, the holding costs and backorder costs until time t are calculated. The size of the customer delivery is also determined. This is done in a way it matches the assumption of delivering every customer order eventually. The customer delivery is placed in *delivery_out*

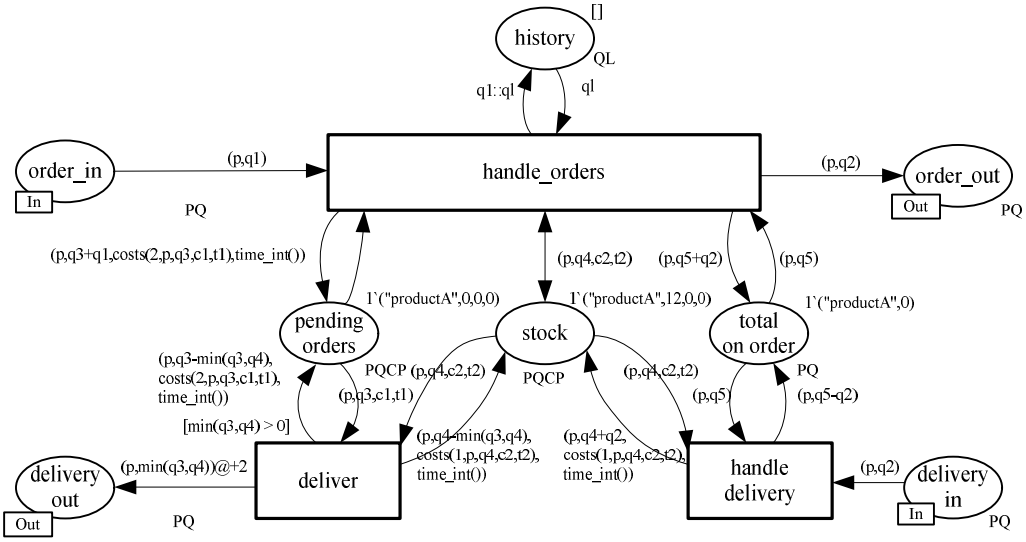

Figure 11 - Sub page component

When a replenishment delivery arrives (in for instance period t), transition ***handle delivery*** is enabled. ***Handle delivery*** adds the replenishment delivery to the stock and updates the holding costs until time t. Also the value of on order is subtracted with the number of received goods.

After ***handle delivery*** has fired, it can happen that transition ***deliver*** is enabled again. It again calculates the holding costs and backorder costs until time t. This is not a problem for the current costs. Because the time interval is equal to zero, both costs remain unchanged. Further a delivery quantity is determined. Because ***deliver*** can fire more than once in a period, this is also the case for the ***handle delivery*** transition. If this is the case, the costs also remain unchanged.

For the retailer, wholesaler and distributor the CPN of figure 11 is used. This sub page can also be used for the manufacturer by adding a transition between ***order_out*** and ***delivery in***.

6.3. Declarations in the CPN model

Declaration	**Explanation**
color E = with e timed;	Unit type
color INT = int;	Integer type
color BOOL = bool;	Boolean type
color STRING = string;	String type
color Product = STRING;	Defines the product type, different products have different names.
color Quantity = INT;	Defines the quantity
color QL= list Quantity;	Defines a list of quantities
color Costs = INT;	Defines the costs. Costs can be holding costs, shipment costs and order costs.
color Period = INT;	Defines the period type
color PQ = product Product * Quantity timed;	Used for the orders and shipments from the one component to the other.
color PQCP = product Product * Quantity * Cost * Period;	Defines the type used for the stocks and backlogs.
var q,q1,q2,q3,q4,q5: Quantity;	Variables used for the quantities
var p:Product;	Variable used to get the correct product.
var ql:QL;	Represents a list that contains the demand

	history.
var c,c1,c2: Costs;	Variables used in cost calculations
var t,t1,t2:Period;	The period until which the costs are calculated.
val delay = 3.0;	Constant, defines the total delay, the order lead-time plus the supply lead-time
val securityFactor = 1.645;	Constant, defines the service level
fun economic_inventory(q4,q3,q5)=q4+q5-q3;	Returns the economic inventory.
fun min(x,y) = if x<y then x else y;	Returns the minimum of two integers
fun max(x,y) = if x>y then x else y;	Returns the maximum of two integers
fun stdevh([],n,sum,sumsq) = Math.sqrt(((n*sumsq) - (sum*sum))/(n*(n-1.0))) | stdevh(q::ql,n,sum,sumsq) = stdevh(ql,n+1.0,sum+real(q),sumsq+(real(q)*real(q)));	Auxiliary function for calculating the standard deviation of a list of integers
fun stdev(ql) = if List.length(ql) <=1 then 0.0 else stdevh(ql,0.0,0.0,0.0);	Given a list of integers, stdev returns the standard deviation of this list of integers.
fun m_a(ql,m,n,sum) = if (ql=[]) orelse (m=n) then real(sum)/real(n) else m_a(tl(ql),m,n+1,sum+hd(ql));	Movavg takes the current demand and the history of demand. From the resulting list the moving average is calculated.
fun moveavg (ql) = m_a(ql,3,0,0);	Returns the moving average of 3 periods
fun moveavg_int (ql) = round(moveavg(ql));	Typecasts the moving average to integer
fun desired_inventory(ql) = round(delay * moveavg(ql) + securityFactor*stdev(ql));	Returns the preferred inventory level used in the third strategy.
fun intmovavg(x,xs) = round(movavg(x,xs));	Typecasts moving average to integer.
fun ordersize(q1,ql,q4,q3,q5)= max(desired_inventory(ql^^[q1])-economic_inventory(q4,q3,q5),0);	Implementation of the third strategy, determines the order size that is placed with the next component upstream.
fun time_int() = IntInf.toInt(time());	Typecasts the current time in CPN Tools from IntInf to Integer type.
fun costs(cpu,q,c,t) = c +(cpu*q*(time_int()-t));	Returns the costs incurred until time t.

Table 6 - Declarations of the Beer Game CPN model

6.4. Turning the model into a game

In the *Beer Game* the player has to decide how much to order from the supplier. The player has no influence in the quantity that is shipped to the customer downstream. The demand is generated in transition *customer* (figure 10). The demand process in this model is deterministic. The demand for the first five periods is equal to 4. In period 6 the demand increases to 8, and will stay at that level for the rest of the game.

Before the game can start the network connection has to be opened. This is modeled in substitution transition *customer*. When transition *connect* has fired, a token is placed in place *connected*, enabling transition generate.

The Beer Game is played for 23 rounds. The end of a game is modeled by enabling the *disconnect* transition after all the demand has occurred. We know when all the demand has occurred. By placing a token with a delay that is higher than the time the last event occurs the network connection is closed after the game ends.

Interaction with the Java application is done in one transition, the ***handle_orders*** transition. This makes the communication between CPN Tools and the Java application easy to implement.

The model was also simulated [A2 for the results] with this demand process. This sudden change in demand had implications for the order sizes seen at the components: the Bullwhip Effect occurred. The ***Beer Game*** was simulated for 23 periods. Also a simulation for 50 periods was run. This is not interesting, because after period 23 the system is in equilibrium again.

6.5. Playing the game

The player plays the role of the retailer. By changing the model, the player can choose another role. This can also be modeled, but it will make the model very complex. It is also possible to play the game with up to four human players. For every player controlled component a network connection can be opened, enabling them to play at the same time. The Java application is implemented to handle this.

Initially the stock positions of the retailer, wholesaler, distributor and manufacturer are set at 12 cases of beer. All the costs are set to 0. The demand for the first three periods is set at 4. Now the beer supply chain is in equilibrium. At each weekly ordering point, you will have to decide how many units to order, based on the following information:

- Your current inventory.
- How much will arrive (on order).
- The size of your most recent order.
- The demand you are currently facing.
- Previous demand you have been unable to meet, and have backlogged.
- The demand forecast for the next period.

7. Conclusion

When a game ends the CPN model belonging to that game is in a certain state. For the CPN model to reach that state, the player has made decisions each round of the game. It can happen that for the same decisions the final state of the CPN model is different. This occurs when certain transitions are enabled at the same time and the CPN Tools scheduler has to decide which transition will fire. To solve this problem the control tokens were added to the *Mortgage Game* and the *Distribution Game*. In the *Beer Game* this is realized by modeling the components such that the total number of goods for that component is the same, regardless of the firing order of the transitions. Only with the *Kanban Game* there were no problems of this kind.

Further the calculation of costs in the games is done using the same cost function. This cost function was first modeled in a colored Petri Net, and then converted to a standard ML function.

Looking back at this internship, I've learned a lot. I had some knowledge of colored Petri Nets and a little of Java. My knowledge of modeling of logistics networks also was not very good. To be able to deliver these results required me to learn a lot. I had no knowledge of CPN Tools, Java programming in general and in JBuilder. Also I didn't know a lot about TCP/IP and how to program networking. Further this assignment made me think more about how to make a decent user interface. A project planning can be found in appendix A5.

I liked the working atmosphere at the IS group very much, also the people there were very helpful when I was stuck in the assignment.

8. References

[1] A simulation game for the service oriented supply chain management
E.G. Anderson, D.J. Morrice, University of Texas, Austin, 2000

[2] Ventana Systems Vensim
http://www.vensim.com

[3] Modeling managerial behavior: Misperceptions of feedback in dynamic decision making
John D. Sterman, Management Science, 35, 3, p321-p339

[4] A simulation model to study the dynamics in a service oriented supply chain
Edward G. Anderson, Douglas J. Morrice, University of Texas, Austin, 2000

[5] Lecture notes Process Modeling, 1BB30
W.M.P. v.d. Aalst, Technische Universiteit Eindhoven

[6] Logistieke technieken
P.P.J. Durlinger, R.P.H.G. Bemelmans, 3rd print, ISBN 90 75325 01 0

[7] Manual KanbanGame
Strategos strategy consultants, www.strategos.com

[8] Fusion places in CPN Tools
http://wiki.daimi.au.dk:8000/cpntools-help/fusion_places.wiki?cmd=get&anchor=Fusion+places

[9] Website of the distribution game
http://www.orie.cornell.edu/~jackson/distgame.html

[10] Website beergame
http://beergame.mit.edu

[11] Bullwhip Effect and supply chain modeling and analysis using CPN Tools
Dragana Makajic-Nikolic, Biljana Panic, Mirko Vujosevic, University of Belgrade

[12] Inventory management and production planning and scheduling
Silver, Pyke and Peterson, ISBN 0 471 11947 4

[13] Comms/CPN: A communication infrastructure for external communication with Design/CPN.
Guy Gallasch, Lars Michael Kristensen, University of South Australia

[14] The Chart2D Java class library sourceforge website
http://chart2d.sourceforge.net

[15] The Java Swing tutorial website
http://java.sun.com/docs/books/tutorial/uiswing/

[16] External communications with Comms/CPN website
http://wiki.daimi.au.dk/cpntools-help/external_communication_wi.wiki?cmd=get&anchor=External+communication+with+Comms/CPN

[17] Definition of the fill rate on wikipedia
http://www.logicjungle.com/wiki/Logistic_engineering

A1. Creating a simple simulation model in CPN Tools

To illustrate how to create a simulation model which has output consider the following example from Lisa Wells. Customers can arrive to ride the Ferris wheel. When a customer arrives, he receives a numbered ticket, and queues up. In the queue the customer waits until he is allowed to ride the wheel. Customers arrive according to a negative exponential distribution (interarrival times are negative exponentially distributed). The Ferris wheel has a capacity of four people. It takes time to load a customer onto the wheel. The load time is normally distributed with a mean of 15 seconds and a deviation of 4 seconds. When the wheel has four customers on it, it will run for a while. The run time is normally distributed with a mean of 300 seconds and a deviation of 30 seconds. If there are less than four customers in the queue, the wheel will also run. When the wheel stops, new customers can be loaded. We are interested in the queue length in time, and the waiting time for each customer. The CPN model can be found in figure 12. One unit on the clock is one second in real life.

First transition *create file* fires. It executes the following code segment:

```
action
TextIO.closeOut(TextIO.openOut "QueueLengthReuzenrad.txt"); // creates the output file
TextIO.closeOut(TextIO.openOut "WT.txt") // creates the output file
```

and places a token in *wait for customer*. Transition *customer arrives* models the arrivals of customers to the system. A customer gets a timestamp and a ticket number. Further the queue length is increased by one, and the customer moves to place *waiting customer*. Further a token is placed with a delay in *wait for customer*, to generate the next arrival of a customer. The following code segment is executed when *customer arrives* fires. The current queue length is increased by one because of the new arrival. The new queue length together with the current model time are written to the output file. Also the new arrival time is calculated.

```
input(q);  // the current queue length
output (nextCust); // to generate the next arrival
action
let
val text=TextIO.openAppend("QueueLengthReuzenrad.txt"); //open the outputfile at the end
in
TextIO.output(text, IntInf.toString(time())); //write current model time to output file
TextIO.output(text, ", ");
TextIO.output(text, Int.toString(q+1)); //write the new queue length to the output file (q+1)
TextIO.output(text, "\n"); // write a newline character to the output file
TextIO.closeOut text; // close the output file
discreteExp(1.0/90.0) // calculate a the delay, which is the new interarrival time
end;
```

Transition *load next customer* handles the loading of the customers. The customers are loaded onto the wheel according to their ticket numbers. The guard of the transition makes sure that no more than four customers are loaded onto the wheel. When a customer is loaded the queue length is decreased by one. The current ticket number is increased by one to get the next ticket. Place *next ticket* models this. Since loading takes time, the token that represents the number of loaded customers is placed in *loaded customers* with a delay. Also a code segment is executed when load next customer fires. This code segment writes the queue length and waiting time to a file.

```
input(q,n,arrivalTime); // current queue length, next ticket, arrival time
output (load Time); // outputs the load time for the customer
action
let
val text=TextIO.openAppend("QueueLengthReuzenrad.txt"); //opens the queuelength output file
val text2=TextIO.openAppend("WT.txt"); // open the waiting time output file
in
TextIO.output(text, IntInf.toString(time())); //convert current model time to string and output it
TextIO.output(text, ", ");
TextIO.output(text, Int.toString(q-1)); // output the new queuelength, decreased because of
service
TextIO.output(text, "\n"); // write a new line to the output stream
TextIO.closeOut text; // close the output file

TextIO.output(text2, Int.toString(n)); //convert the current ticket number to string and output it
TextIO.output(text2, ", ");
TextIO.output(text2, Int.toString(inttime() - arrivalTime)); //calculate waiting time for customer
TextIO.output(text2, "\n"); //write a newline character to the output stream
TextIO.closeOut text2; // close the output file

discreteNormal(15.0,4.0) // calculate the load time and bind it to the output arc

end;
```

Transition ***start wheel*** fires when the wheel is fully loaded, or when there are less than four customers loaded onto the wheel. The situation can happen when the customer queue contained less than four customers at some point in time. The guard of transition ***start wheel*** models this. The run time of the wheel is normally distributed with a mean of 300 seconds and a deviation of 30 seconds. This is handled in the code segment that is executed when ***start wheel*** fires. The following code segment belongs to ***start wheel***. ***Start wheel*** places a unit token with delay run time in place ***running***.

```
output (runTime); // bind the result to the outgoing arc
action
discreteNormal(300.0,30.0); //calculate run time
```

Finally transition ***stop wheel*** handles the unloading of the customers by putting a token with value 0 in place ***loaded customers***.

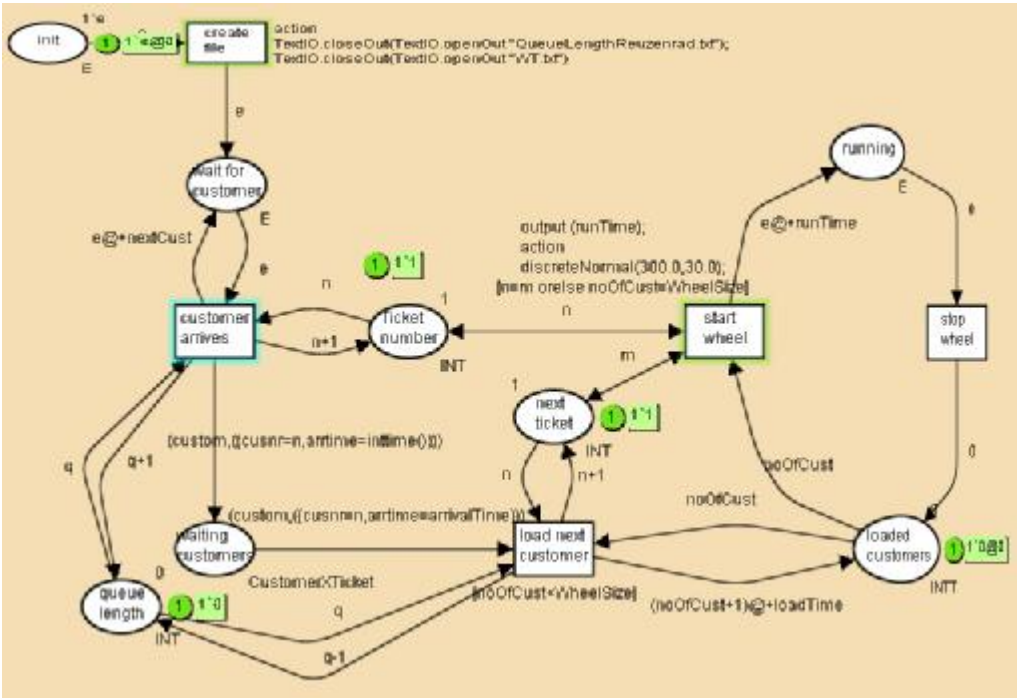

Figure 12 - Ferris wheel CPN model

The output files are in comma separated value format, and can be used in a spreadsheet program. Three situations where simulated. Arrival rates of 1 customer per 60 seconds, per 90 second and per 120 seconds. Queue lengths and waiting times were measured. Only one run for each situation was simulated. The results can be found in the following graphs.

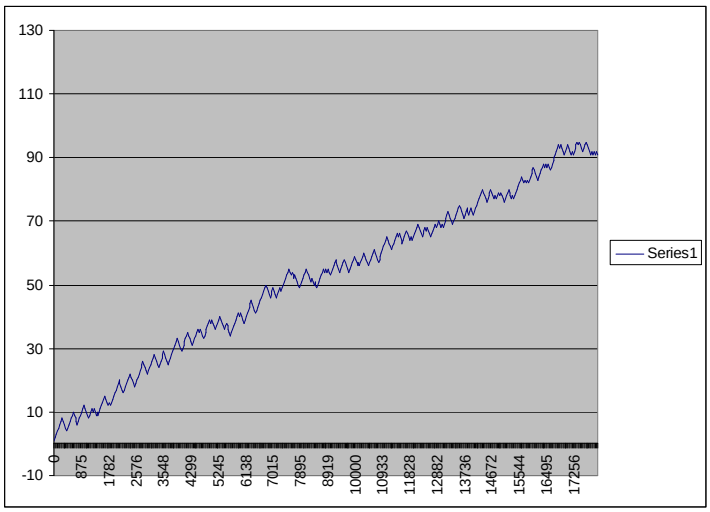

Figure 13 - Queue length, λ=60.0 seconds

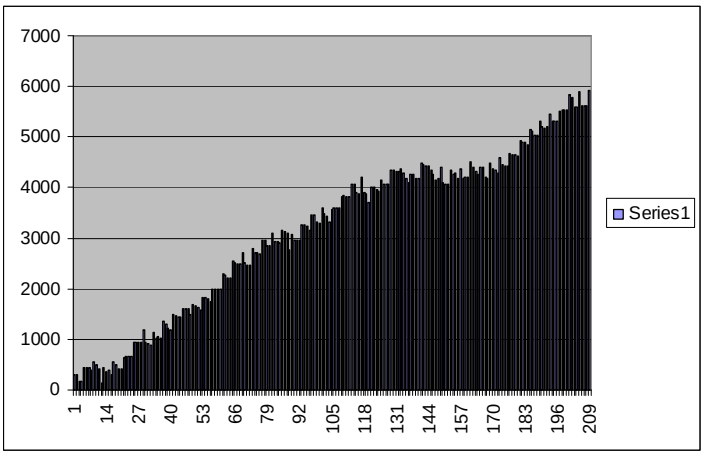

Figure 14 - Waiting time per customer, λ=60.0 seconds

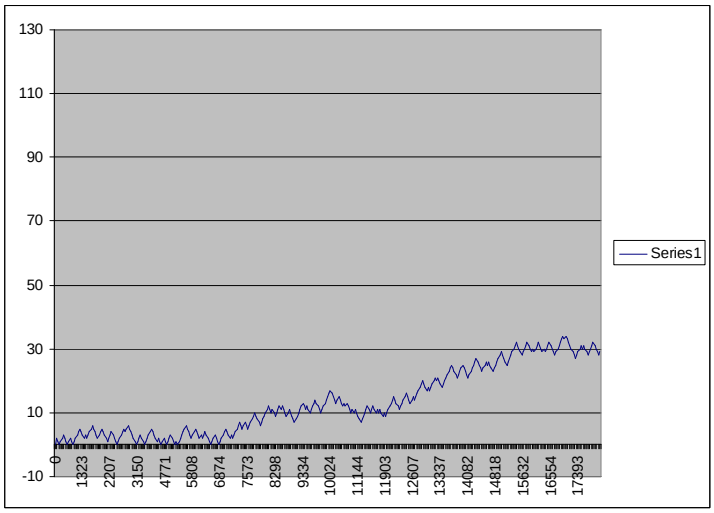

Figure 15 - Queue length, λ=90.0 seconds

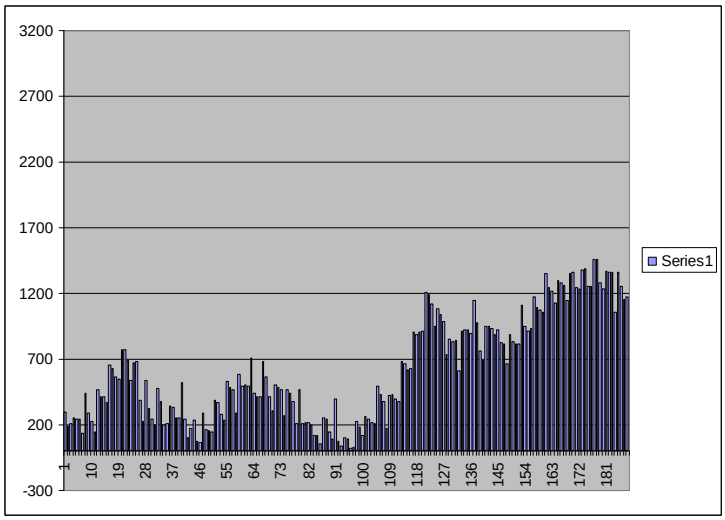

Figure 16 - Waiting time per customer, λ=90.0 seconds

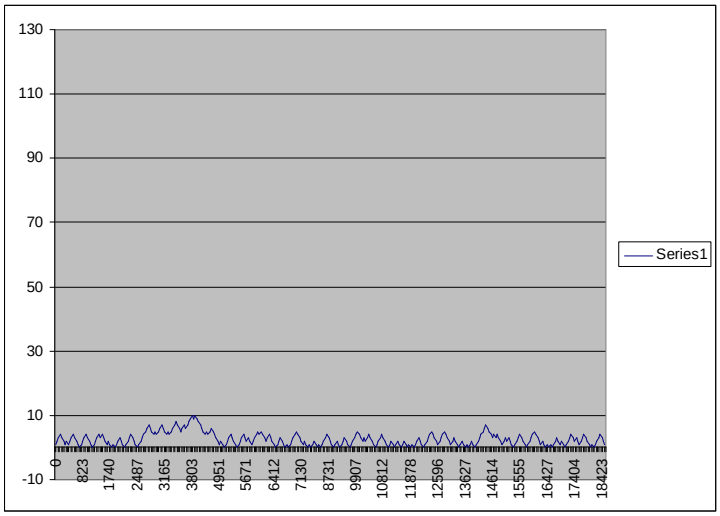

Figure 17 - Queue length, λ=120.0 seconds

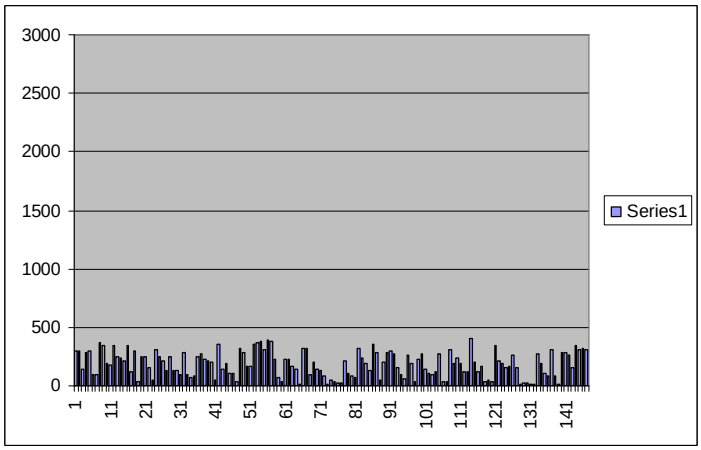

Figure 18 - Waiting time per customer, λ=120.0 seconds

For simulating the CPN model the simulation toolbox of CPN Tools can be used. By using the fast forward button the system advances 50 steps. However, when simulating a real life situation it is more intuitive to run the simulation for a certain amount of time units. This is realized using the following ML code, which can also be found on the CPN Tools mailing list.

```
CPN'Options.stop_crits := [CPN'Options.until_time (Time.fromInt (x))];
```

The value for x is the number of time units the simulation runs. The simulation cannot be performed using the standard simulation tool box of CPN Tools, because it resets the criteria set by the user. Instead the simulation has to be run by executing the following ML code.

```
val (stepstr, timestr, resstr) = CPN'Sim.run();
```

Executing ML code is done by selecting the ML button in the simulation tool box and the clicking on the ML code that needs to be executed.

In modeling there can be improved a lot. How to separate the CPN model from the measurement system for instance. CPN Tools has the possibility for fusion places. These are places that act the same as a global variable and probably can be used for this. Also in simulating you have to simulate a couple of runs. This is possible in CPN Tools, but very hard to program. This is because the functions necessary for this aren't documented at all. There is

a CPN model available which has support for multiple runs. The code that handles the simulation isn't properly documented. It uses methods from CPN Tools itself. The code is a prototype of what is going to be implemented in CPN Tools.

A2. Simulation results for the Beer Game

The graphs below show the results of the simulation model of the beer game. In the first graph the bullwhip effect is visible. Only the data for the first 24 periods is in the graphs. After the 24th period the system settles in equilibrium again.

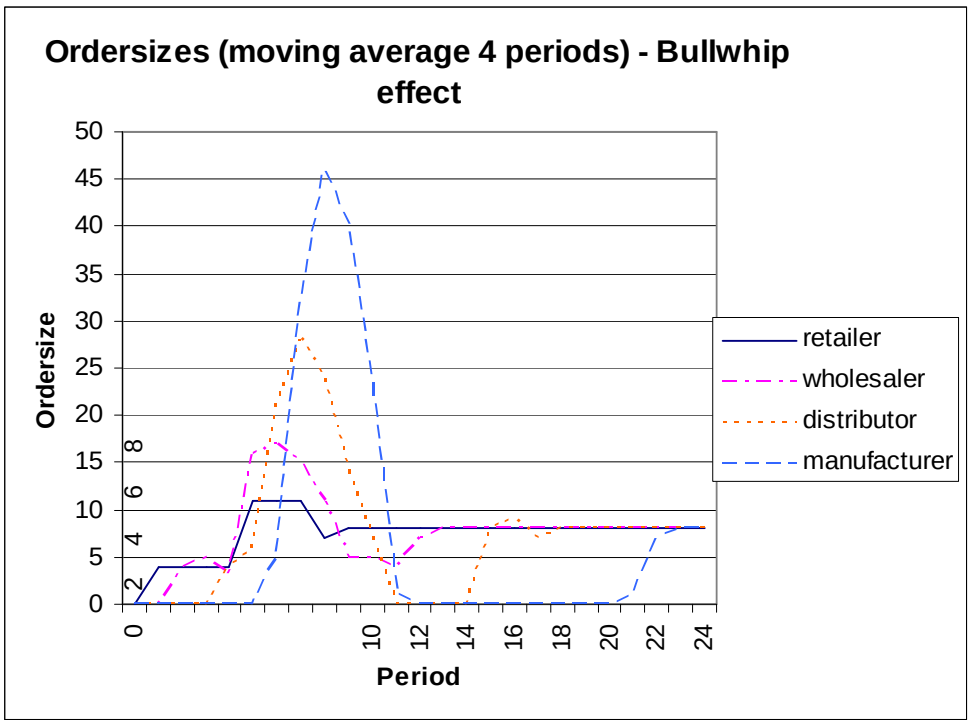

Figure 19 - Order sizes

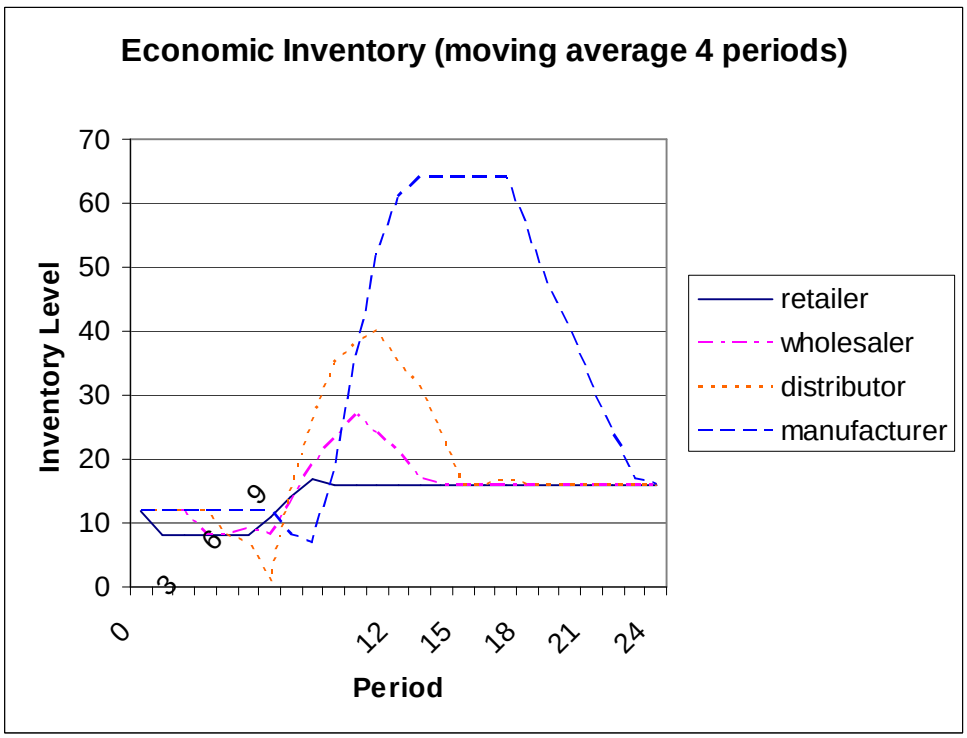

Figure 20 - Economic inventory positions

These charts can be obtained by using the method described in Appendix A1. The following code segment has to be added to the *handle_orders* transition in the *Beer Game* simulation model:

```
input(b,c,d)
action
let
val st = TextIO.openAppend("economic_inv_component.txt");
in
TextIO.output(st,Int.toString(tijd()));
TextIO.output(st,",");
TextIO.output(st,Int.toString(economicInventory(b,c,d)));
TextIO.output(st,"\n");
TextIO.closeOut st
end;
```

The input variable b is the *stock* level, c are the *pending orders* and d is the *total on order* quantity. The current time and the economic inventory level for that current time are written to a text file. This text file can be loaded into a spreadsheet program to draw the chart.

A3. How to start the games

To run the games you need to have java run environment version 1.4.1_02-b06 installed. Further you need to have at least CPN Tools version 1.1.8 installed, but version 1.2.0 is recommended.

For all four games to run you need to do the following:

1. Open the model in CPN Tools
2. Open the simulation toolbox and set the number of steps to a large number, for instance 1000000. When you are using CPN Tools version 1.2.0, you will also need to set the time between firings to 1 millisecond.
3. Press the play button.
4. Double click on the Java applet that belongs to the model. The game will only work if you completed steps 1 to 3.
5. Play the game.

A4. Project Planning

Introduction

As part of the curriculum of the computer science program at Eindhoven University of Technology (TU/e), students have to do an internship. Typically one term is reserved for the internship. One term corresponds to 14 weeks. This is the project plan for the project I'm doing at the information systems (IS) department at the faculty of technology management (TM) at TU/e. The project plan will contain the following:

context of the project Description of the starting position, context and background situation of the project.

project description Description of the project itself, the goal and the expected outcome.

parties involved and organization Description of how the parties are involved in the project and how the project is organized.

planning Description of how many time is spent on different parts of the project and how long it will take to finish the complete project.

risk analysis An analysis of risks that apply to this project and if necessary a formulation of countermeasures for problems or risks that are likely to occur.

Context of the project and starting position

The starting point for this project is a paper submitted to the CPN workshop about the modeling of a supply chain in CPN Tools. The paper describes a way of how to model the beer game in CPN Tools. However this model is very simple and is more an introduction of how to model supply chains in CPN Tools. Further the lecture notes of the course process modeling (1BB30) are available. These provide a good introduction to CPN Tools and the ML language. Also the rules of the beer game are available. Documentation for the ML language as well as CPN Tools can be found on the internet.

Project description

As stated above the project description consists of a description of the project itself, what are the goals and what is the expected outcome.

The project consists of: improving the model found in the paper submitted to the CPN workshop. The improving is about adding a simulation environment and finding a way to write the result to an output file so that it can be used in for example a spreadsheet program. Other forms of IO also have to be considered.

For the simulation environment the statistical library of CPN Tools has to be used, and for writing output to a file code segments have to be used. Looking into these is also part of the project. The statistical library contains functions for probability distributions, for example the negative exponential distribution which is used a lot in simulation. Code segments are small pieces of ML code that are executed once the transition the code segment belongs to fires.

Further a list of logistic games along with their description has to be made. Next a selection of games has to be made from this list.

The expected outcome is a couple of logistic games modeled as simulation models in CPN Tools, which can be run immediately. These models are turned into games by programming a Java application for each of the games. The models together with their JAVA applications will be put on a website. This website also has to be created. Further a report will be made which will contain information of how everything is put together.

Parties involved and organization

The student that will be working on this assignment is Marcel Welters. He will be supervised by a supervisor from TU/e. The supervisor is Marc Voorhoeve of the information systems (IS) department of the faculty of mathematics and computer science. The project is provided by the IS department of the technology management faculty. Therefore it will be carried out there as well and he student will also have a supervisor there. The supervisor of the IS group is Prof. Dr. ir. Wil van der Aalst. The creator of colored Petri nets, Prof. Kurt Jensen will be visiting the TU/e. So at some point there will be interaction between the student and Prof. Kurt Jensen, about the assignment and CPN Tools.

The Information Systems sub department (IS), formerly known as the Information and Technology sub department (I&T), operates in a domain that is subject to constant and rapid change. Both the technology and its applications change at a high pace. The IS embeds three research groups. One of these is Business Process Management (BPM). This subgroup best matches the domain of the project as the focus of this research group is on the modeling and analysis of operational business processes. The goal of the resulting models is either to gain insights in the processes to be supported or to specify and/or configure information systems. Keywords: process models, data/object models, UML, Petri nets, data mining, performance analysis, validation, verification, simulation.

Further is can happen that a some point some advise is needed from the people of the logistics group of the faculty of technology management of which logistic games are interesting to model and which not.

Planning

Project start date: September 1, 2004
Project end date: December 1, 2004

The internship has duration of 14 weeks. The student has to spend 40 hour per week on the internship, making it a total of 560 hours. The 14 weeks include working on the report and if necessary preparing a presentation of the results.

Week	*Activity*
36	Creating a project plan
37	Research on logistic games and CPN Tools
38	Research on statistical library and simulation environment
39	Research on ML language and code segments
40	Model beer game in CPN Tools (weeks 37-40: small iterations)
41	Select games that are going to be modeled
42	Finalize beer game model and start with Comms/CPN
43	Comms/CPN
44	More java, look at the other games/CPN models
45	More java, look at the other games/CPN models
46	More Java, add visualization to GUI
47	Programming applications
48	Programming applications
49	Programming applications
50	Programming applications
51	Report and website, finalizing applications and models
52	Report and website, finalizing applications and models
53	Report and website, finalizing applications and models
1	Report and website, finalizing applications and models

This planning is a total of 14 weeks. The planning changed after meetings with the supervisors. Five weeks were added. The total duration of the project was 19 weeks.

Risk analysis

Because this is an internship, things can go wrong (they probably will). A risk analysis is conducted to list for example some common risks together with the appropriate action if one would occur. Also the probability of the event actually happening has to be mentioned. These probabilities have to be guessed and therefore I've left them out.

more work than expected It is always possible that the planning isn't correct and that some activities take more time than expected.

unclear goals and expectations To have clear goals and clear mutual expectations is very important for the success of a project. This document lists the goals and expectations.

bad communication Lots of projects fail due to bad communication. The student will report to the supervisors at least once a week. When the project progresses this can probably be reduced to at most once a week.

work place The IS department of TM provides the work place for the student. In case there's no computer available the student will work at home.

References

[1] http://www.jjjkruisprojectwerk.nl/bedrijfskundigeinvalshoek/projectmanagement/projectplan/index.html
Project plan example

[2] Opleidings gids technische informatica 2004-2005

[3] http://www.tm.tue.nl/capaciteitsgroep/it/
Website Information Systems group of the technology management department

A5. Screenshots of the Java applications

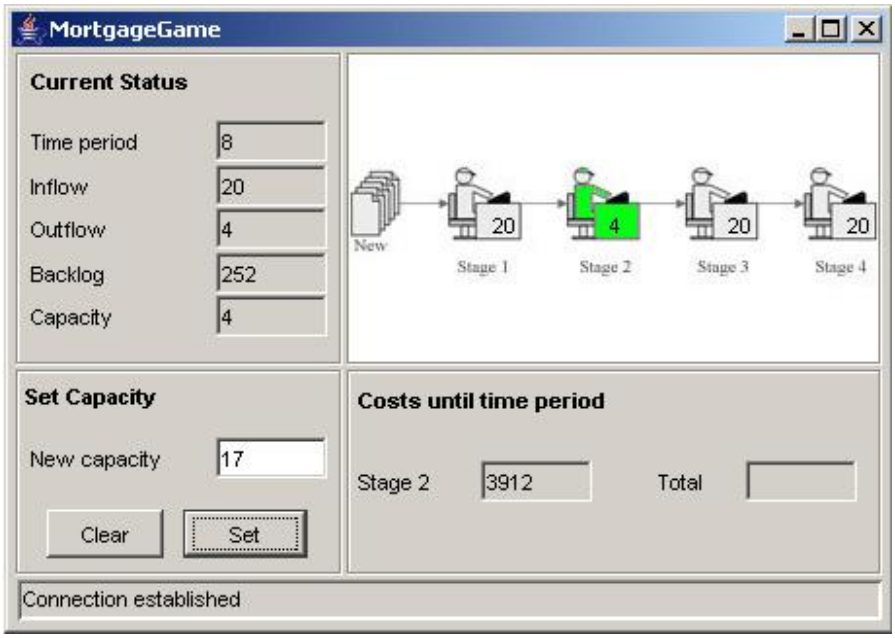

Figure 21 - Mortgage game

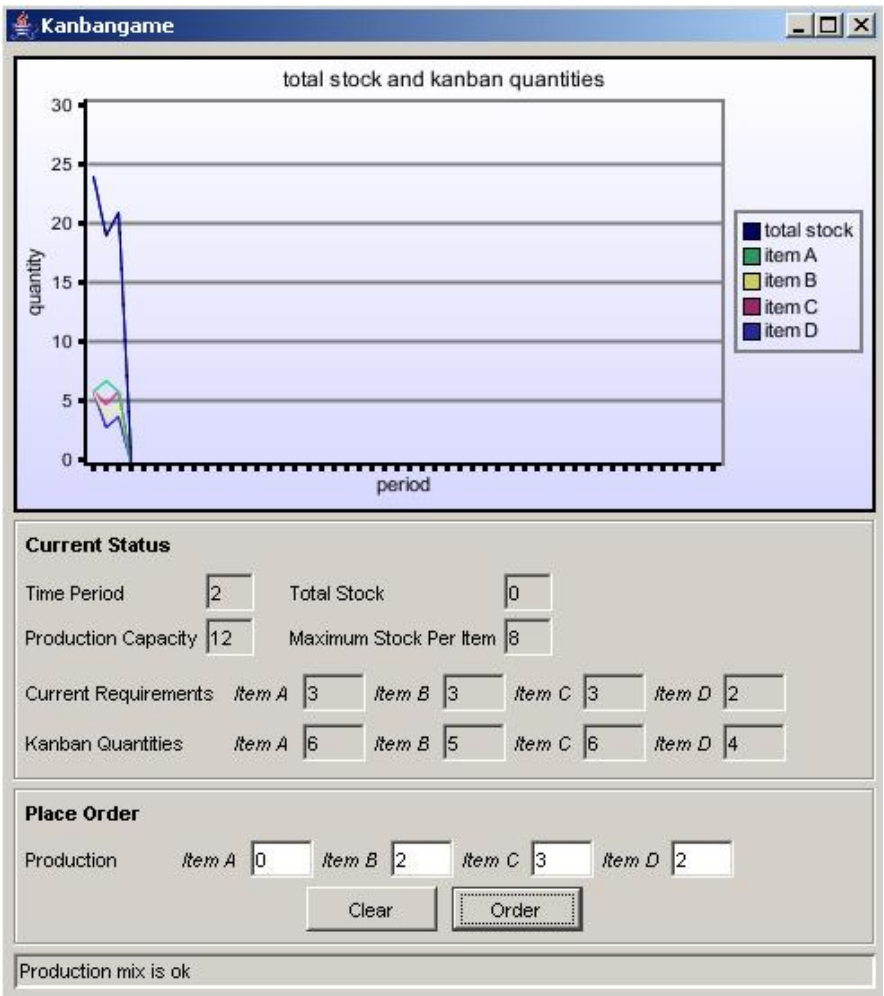

Figure 22 - Kanban game

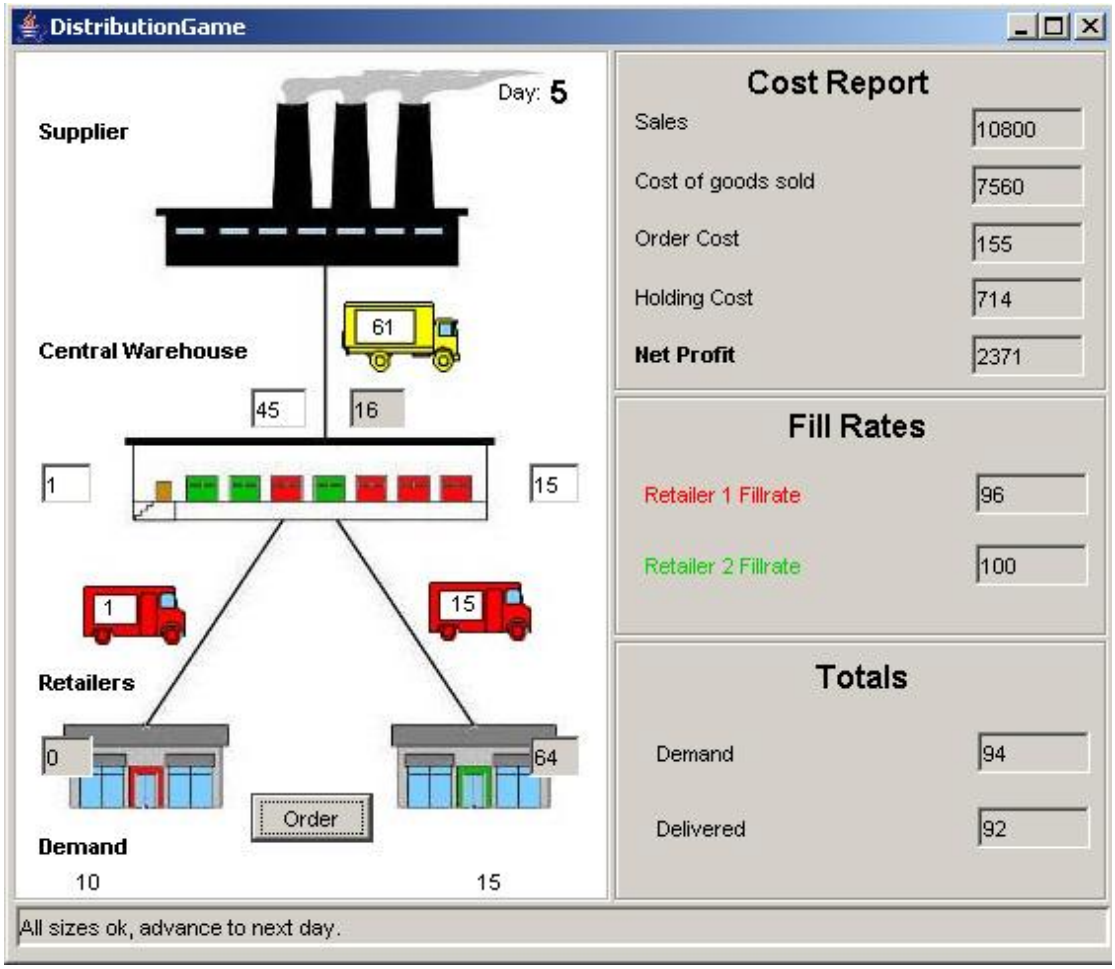

Figure 23 - Distribution game

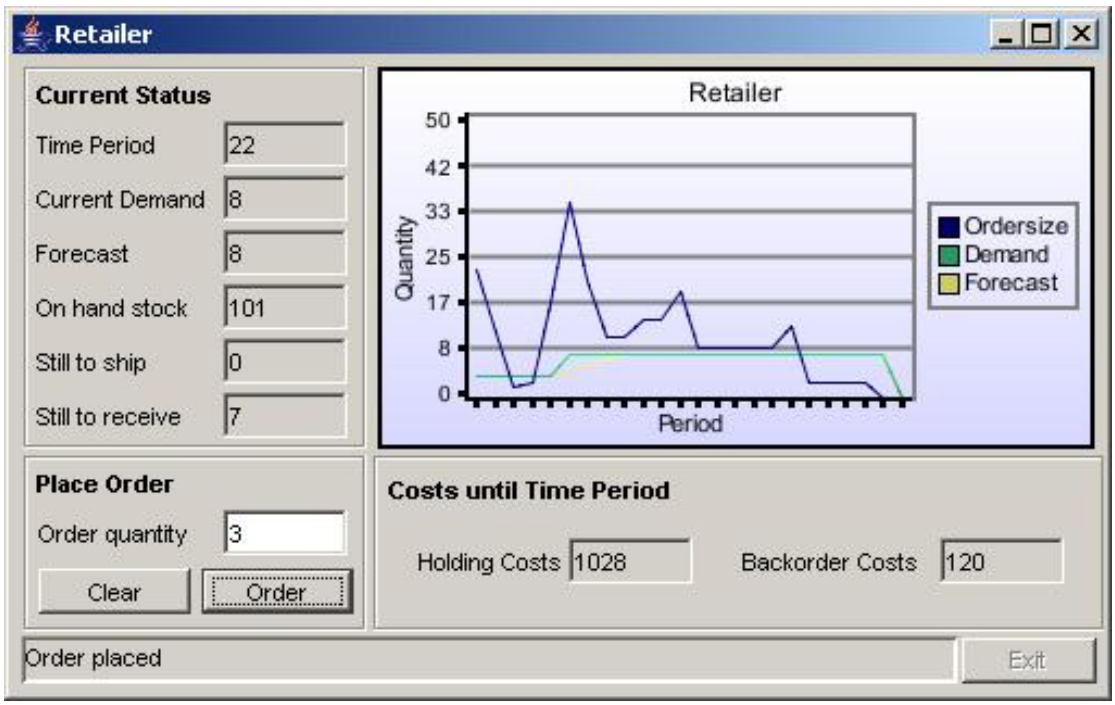

Figure 24 - Beer game