

Tray allocation for a sortation system II

Model driven development using VDM++ and UML II

Spring 2010



Kim Bjerge kbe@iha.dk 20097553

José Antonio Esparza jaesparza@gmail.com 20097706

Table of contents

1. Introduction	3
1.1. Report structure	3
2. System description	5
2.1. Purpose of the models	5
3. Concurrent model	7
3.1. UML model	7
3.1.1. UML class and object diagrams	7
3.1.2. Sequence diagram	9
3.2. The VDM model	12
3.2.1. The World class	12
3.2.2. The sorter environment class	12
3.2.3. The tray step class	14
3.2.4. The induction controller class	15
3.2.5. The tray allocator class	16
4. Real-time Distributed model	20
4.1. System deployment	20
4.1.1. Scenario 1	21
4.1.2. Scenario 2	23
4.2. The real time model in VDM++	25
4.2.1. The SorterEnviroment class	25
4.2.2. The InductionController class	26
4.2.3. The induction class	29
4.2.4. The tray allocator class	29
5. Test setup	32
5.1. Concurrent version setup	32
5.2. Real time version setup	35
6. Proof obligation and Test coverage	44
6.1. Proof obligation	44
6.2. Real Time version test coverage summary	45
7. Conclusions	47
8. References	49

Appendix A Test Result

Appendix B VDM Concurrent Model

Appendix C VDM Real-time Distributed Model

1. Introduction

From the early ages of computing, computer systems have always work interconnected and often with shared resources. This situation has continued until nowadays and a computer cannot be conceived as an isolated resource. In this scenario concurrency and real time requirements have to be carefully considered. Appropriate methods should be used in order to avoid complicated issues, such as deadlocks, debugging and fault detection in this kind of distributed systems.

As with many other situations, creating a model of the system under study can be extremely helpful to cope with constraints or avoid design errors like bottlenecks. By using formal methods in modelling and tools that support real time analysis these situations can be avoided from the early stages of the development process. A possible language to manage concurrent scenarios from the formal method perspective is VDM++, which in combination with environments like overture or VDMtools can be used to perform advanced concurrent and distributed real time analysis of a model.

Furthermore, in the case of highly complex systems, other tools like petri nets and coloured petri nets¹ can be used for validation and modelling of distributed systems, but their application are out of scope of this project.

In order to work with concurrent and real time constraints, a scenario introduced in the report Tray allocation for a sortation system, presented in the course VDM1 will be modified. The introduced modifications will bring the previous sequential model focused on functionality to two different models the concurrent and the distributed real time model. Both of them will be discussed and explained in this report.

1.1. Report structure

The document is composed by seven main parts, in which both concurrent, real time (RT) and distributed models are discussed. The document starts with the introduction, in which the project is presented and an overall overview of the present work is shown. The chapter system description, introduces the system and the relevant abstractions that have been used in the models. The concurrent model will be presented, going through UML class and sequence diagrams and finally through the VDM model. The chapter that describes the real time and distributed model will introduce the model created based upon the concurrent model, focusing on the usage of periodic threads and deployment to different CPUs. The chapter test setup will expose how the present models are tested and describe in more details the process coming to the final RT model. In the chapter proof obligation and test coverage, the degree to which the

¹ Coloured Petri nets conform a mathematical modeling language for description of distributed systems. More information can be found at reference [6]

internal model quality has been tested will be presented. Finally, some conclusions and thoughts will be exposed in the conclusion section. Additional references and consulted material will be described in the References section.

2. System description

The present project is based on the system introduced in the report “Tray allocation for a sortation system”, analyzed and modelled in the previous VDM course. A brief description of the system will be given at this point, nevertheless in order to consult further details about the requirements or the normal system operation, sections 2.2 and 2.5 from the previous report should be consulted [1].

The simplified sortation system under study is composed by three inductions responsible for inducing items on to a sorter ring, a moving belt in charge of transportation. The inductions are fed by external actors, may be at a constant or inconstant rate. The inductions are grouped in induction groups. Before every induction group a card reader is located to determine the id of the tray in front of it. A simple representation of the system can be seen in the figure 1.

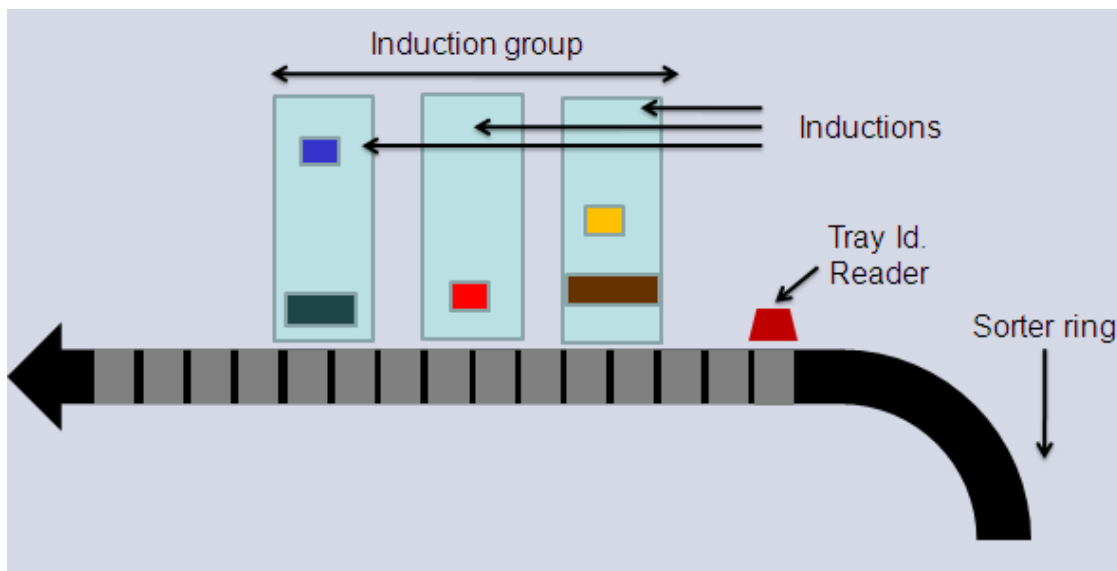


Figure 1: A simplified sorter model

2.1. Purpose of the models

In the previous project the functional part of the model was covered, by creating a strategy to allocate properly the items on to the sorter with no starvation at the inductions. In this project, we will focus on modelling the different concurrent actors that are involved in the system, and regulating the way they interact. This concurrent model allows separating the functionality and emphasizes the logical restrictions that influence the communication. By creating such a model, a more realistic approach is reached, because in the current real time system several actors are performing actions simultaneously, and not in the sequential manner that was previously described.

The purpose of creating a real time and distributed version of the concurrent model is to be able to deal with constraints that are inherent to the physical system on where the model will be deployed. By doing so, a more accurate and meaningful approach will be obtained, being

able at this point to represent the communication between CPUs in the system and the mechanical movement of the sorter. The mechanical movement of sorter will be modelled given by the time it takes a tray based on its size to be moved one step at a given sorter speed.

At the same time letting the environment be feeding items to the inductions precisely and synchronized in time and simulation time. Furthermore, valuable information related to the system and its performance can be obtained, as well as design requirements regarding to bus bandwidth or CPU speed.

3. Concurrent model

In the current chapter, the concurrent version of the system will be presented from both UML and VDM++ perspective.

3.1. UML model

The UML model of the system has been created by using class and sequence diagrams. Since the main entities that conform the classes, and its methods and attributes were already detected in previous work (chapter 5 at [1]), we have focused on analyzing which of these classes should represent entities that behave concurrently.

3.1.1. UML class and object diagrams

One of the changes that have been introduced in the model is the use of two different time units in the system. This change is realized by the existence of the classes TimeStamp and TrayStep. The TimeStamp class is described in the document "Development Process of Distributed Embedded Systems using VDM" see [9] chapter 3.6.6. The TimeStamp class is responsible for updating the *current time* (measured in milliseconds) and is associated to the world class. The SorterEnvironment thread updates the current time by use of the TimeStamp class which is synchronized with the TrayStep class that provides functionality to calculate and synchronize the time it takes to move one tray based on the current time provided by the TimeStamp class. The TrayStep class will maintain equivalence between the time and a physical tray step in the sorter. These classes introduces an important separation between the physical notion of time and how it is affecting the overall system. They are used to synchronize simulation and execution of the InductionController and TrayAllocator threads. The system threads will now be operating in parallel synchronized by the new classes TimeStamp and TrayStep.

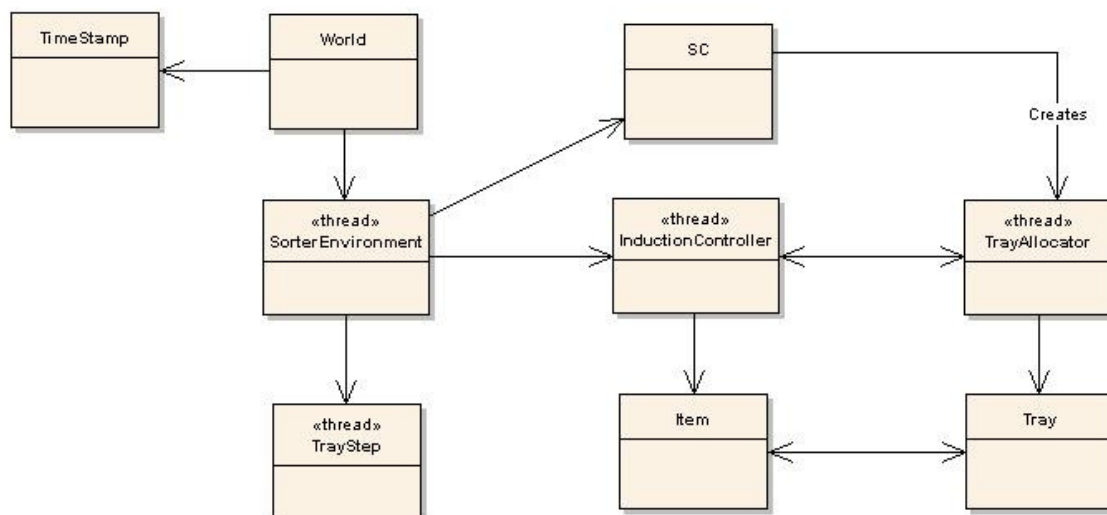


Figure 2: UML class diagram for the concurrent model

In figure 3, an active object diagram for the current system is introduced. This gives a closer view to the system, showing that in this case three induction controllers are running simultaneously in separated threads, connected with the sorter environment and the tray allocator classe. In order to illustrate the system from a more clear perspective the item and the tray classes are not shown.

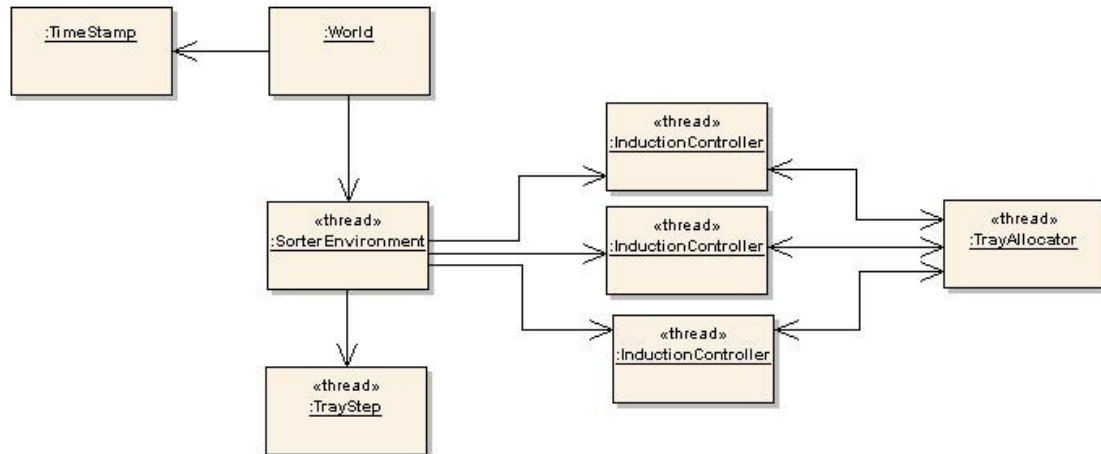


Figure 3: UML Object diagram for the concurrent model

3.1.2. Sequence diagram

In this chapter we will describe the new interactions between classes and threads in the concurrent model compared with the previous sequential model.

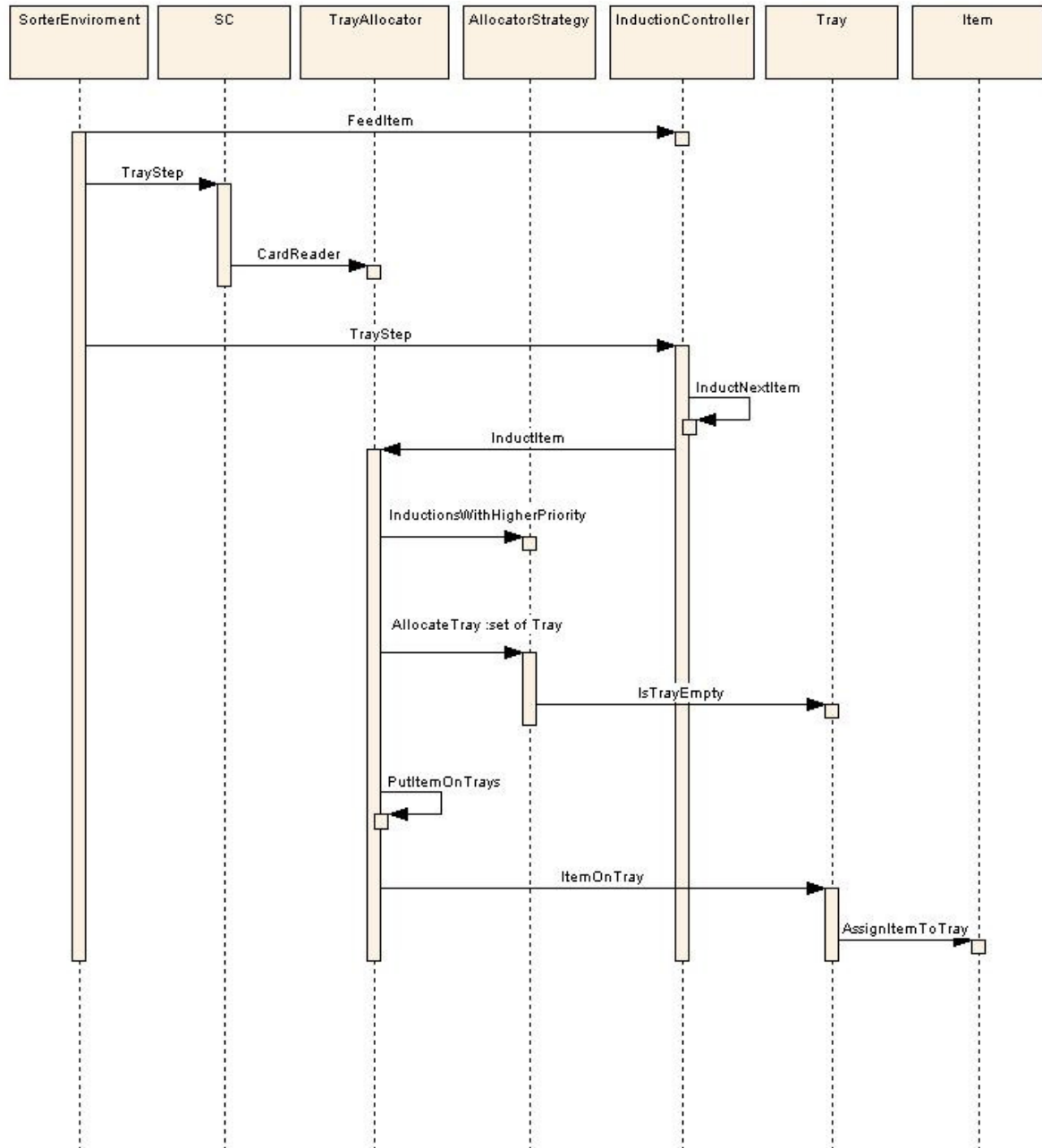
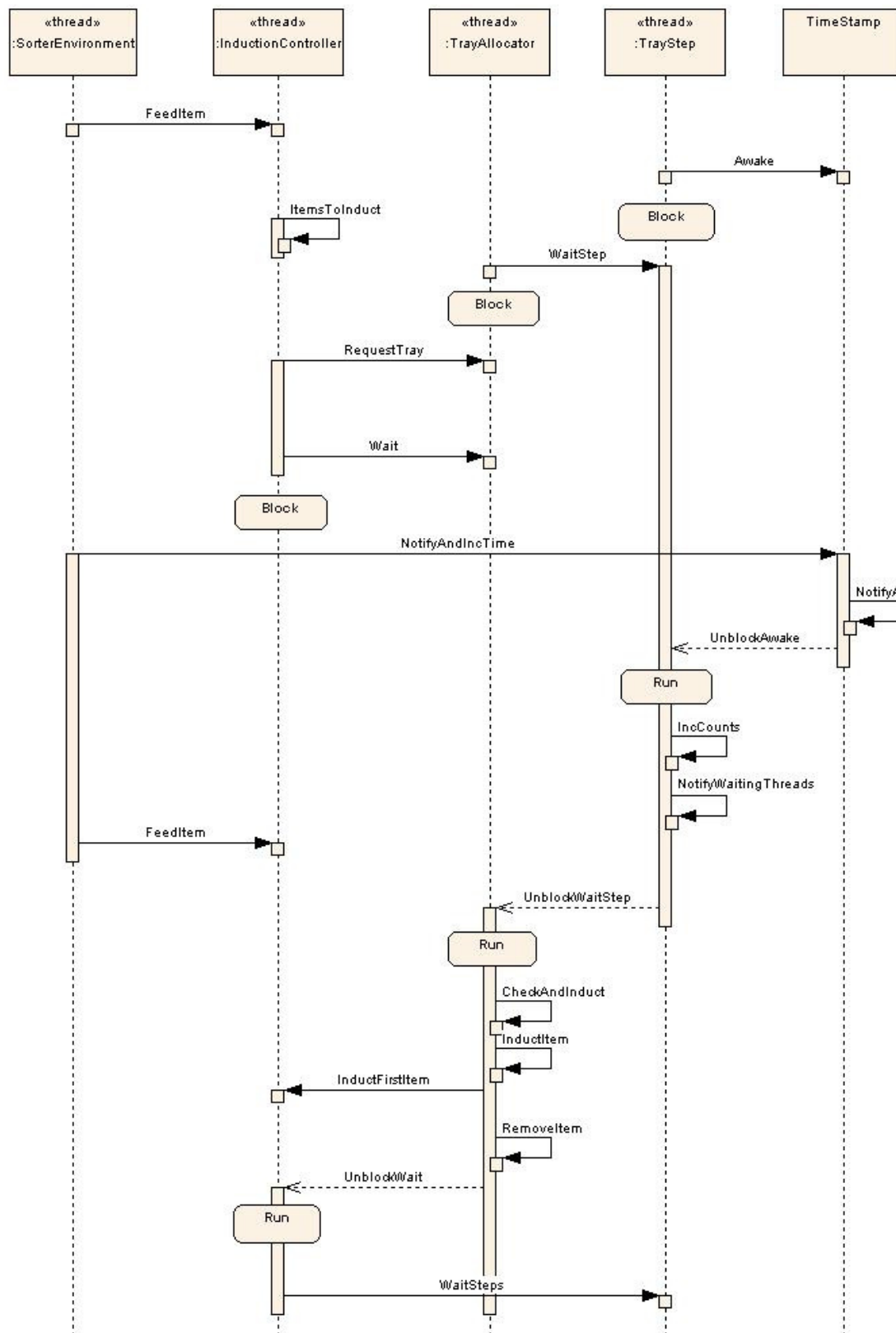


Figure 4: UML Sequence diagram for sequential model

If we compare the sequence diagrams for the sequential and concurrent VDM model we have now changed the control for the InductionController (IC) and the TrayAllocator (TA) in making it possible for them to operate independently of each other.

**Figure 5: UML Sequence diagram for concurrent model**

If we take a closer look at the sequence diagram for the concurrent model we have the `SorterEnvironment` that feeds items to the IC. The IC thread is blocked on waiting for

ItemsToInduct and performs a RequestTray at the TA where it will be blocked until the TA is unblocked waiting for the next tray step synchronized by the TrayStep thread. While the IC thread is waiting it is possible that new items could be feed by the SorterEnvironment. The logic for decision on when to induct an item is now moved to the TA. It will check if any inductions are waiting with items to be inducted by calling the operation CheckAndInduct. For every IC waiting with an item it will induct the items by using the operations we already have tested in the sequential model (InductItem using the strategy pattern). Finally when an item is inducted the blocked IC waiting with an item will be release to induct the next item. Before the IC repeats inducting the next item it will have to wait the induction separation time therefore it calls the WaitStep operation in the TrayStep class.

In the following chapter the VDM++ model that realizes the above concurrent UML model will be explained in details.

3.2. The VDM model

3.2.1. The World class

The world class is responsible for associate a time reference provided by the TimeStamp class to the sorter environment associated to itself. This time reference and its construction can be seen in the following model snippet.

```
public static timerRef :TimeStamp := new TimeStamp();
```

The world class is responsible as well for initiating the whole simulation based on the contents of the simulation file and present through the console of the final results.

3.2.2. The sorter environment class

Starting Threads

The sorter environment class is responsible for initialize the TrayStep class. This class makes it possible to generate one tray movement depending on the current time updated by the TimeStamp class. Since it is updating an internal tray step counter it should be run concurrently and it is started in a new thread as follows:

```
busy : bool := true;
public static trayStep : TrayStep := new TrayStep(Speed);
```

In the first line it is possible to see a boolean variable named "busy", this will help to deal with when to stop the debug thread for it to print the final simulation results.

```
public isFinished : () ==> ()
isFinished() == skip;
```

The isFinished operation is not performing additional actions, because it is just composed by a skip statement. The isFinished operation is used in combination with the busy variable and the permission predicate defined in the sync primitive. The debug thread in the world class will be block on this operation until the environment sets busy to false, indicating that the simulation is finished.

```
sync
    per isFinished => not busy;
```

The sorter environment will start the different threads for the trayStep and allocator. It is taking responsibility as well for starting the different inductions contained in the induction group. This is done by using the quantifier "forall" and going through all the elements in the set composed by the number of inductions. It iterates through all the inductions contained in the induction group and starts them on separated threads.

```

thread
(
    sc.CreateAssignments(World`env);
    start (trayStep);
    start (SC`allocator);
    forall i in set {1,...,TrayAllocator`NumOfInductions}
    do start (inductionGroup(i));

```

Feed inductions

The sorter environment is feeding the inductions with items during its execution. This is illustrated in the next model snippet.

The sorter environment will go through all the inductions contained in the sequence induction group. For each induction it will try to retrieve an item from the itemLoader, by executing the operation `GetItemAtTimeStep`, the execution result will be stored in the variable `size` defined in the `let` statement. The `GetItemAtTimeStep` will now return items at time step in ms instead of tray steps as used for the sequential model. In the case an item is found (`size < 0`) a new instance of the item will be created and it will be fed to the proper induction.

```

while busy do
(
    for all i in set {1,...,TrayAllocator`NumOfInductions}
    do
    (
        let size = itemLoader.GetItemAtTimeStep
                                (World`timerRef.GetTime(), i)

        in
        if (size > 0)
            then
            (
                itemId := itemId + 1;
                inductionGroup(i).FeedItem(new
                                                Item(size, itemId));
            )
    );
);

```

Advance time

The sorter environment is responsible for controlling the time signal provided by the timestamp class associated to the World. The environment is the only thread that is executing the `NotifyAndIncTime` operation, which is incrementing the current time counter (Steps of 10 ms) and notifying the rest of the running threads.

In the case the current time maintained by the TimeStamp class is greater or equal to the simulation time defined in milliseconds in the scenario test file, the busy variable is set to false. This will mean that the simulation has finished.

```
World`timerRef.WaitRelative(0);
World`timerRef.NotifyAndIncTime();
World`timerRef.Awake();
if (World`timerRef.GetTime() >= itemLoader.GetNumTimeSteps()) then
    busy := false;
);
```

3.2.3. The tray step class

The TrayStep class defines a second time unit that is used to synchronize tray steps in the system. This class maintains equivalence between the TimeStamp class and the required amount of time to produce a tray step to the system. In order to maintain this equivalence and synchronization of the system the function WaitSteps is used. This function will force the TrayStep to wait until current time has reached the time for a new tray step. This mechanism is modelled by use of a map, which will contain the associated thread id. This is the same way as the TimeStamp class is modelled as described in [9] chapter 3.6.6

```
Public WaitSteps : nat ==> ()
WaitSteps(steps) ==
(
    AddToMap(threadid, counts + steps);
    Wait();
);
```

Once the thread has been added to the waiting map, the wait operation is called. This operation will be blocked until the treadid is removed from the waitTrayStepMap.

```
Wait: () ==> ()
Wait() == skip;
```

This synchronization mechanism is used in order to handle concurrency. The following permission predicate on the wait operation can only be executed in the case the invoker thread is not already in the waitTrayStepMap.

```
sync
per Wait => threadid not in set dom waitTrayStepMap;
```

The operations performed by the TrayStep thread are illustrated in the following model snippet. The tray step class will increment the tray step counter. Waiting threads will be notified. It will

then wait the amount of milliseconds that are required to generate the next step. Once this time has elapsed, the rest of the threads are notified.

```
thread
while true do
(
    IncCounts();
    NotifyWaitingThreads();
    World`timerRef.WaitRelative(stepTime);
    World`timerRef.NotifyAll();
    World`timerRef.Awake();
)
```

3.2.4. The induction controller class

Each induction controller is responsible for managing a single induction and is associated with the sorter environment and the tray allocator. Each induction has an associated induction controller running in a separate thread. Since they are constantly managing items received from the environment and putting them on the sorter, the concurrency should be carefully managed. Otherwise, not regulated concurrent actions over shared data could lead to a nondeterministic situation, causing system instability.

This concurrency problem is addressed by the use of mutexes. They are shown in the following model snippet. It is important to keep mutual exclusion between the `FeedItem` operation and the `InductFirstItem`, since both of them are acting over the sequence of items contained at the induction and called by the environment and induction threads.

The execution of `InductFirstItem` and `IncrementPriority` should be done individually. That means that it should not be possible for two threads to access the same induction controller instance and execute `InductFirstItem` or `IncrementPriority`. In order to prevent this situation, the mutexes over these operations have been added, even though in the current model it is just one thread in the `TrayAllocator` that has access to them.

The operations `IncrementPriority`, `InductFirstItem` and `GetFirstItem` should not be executed simultaneously, since the last two are working with the same sequence of items. Finally if there are no items to induct, the thread will be blocked. Since the items are contained in the sequence `Items`, it is enough to check its length.

```

sync

    mutex (FeedItem, InductFirstItem);
    mutex (InductFirstItem);
    mutex (IncrementPriority);
    mutex (GetFirstItem, IncrementPriority, InductFirstItem);
    per ItemsToInduct => len items > 0;

```

The following model details the sequence of operations that are performed by the active thread of the induction controller. While there are items present in the induction, the induction controller will get the first item from the induction and request a tray at the tray allocator. The induction will wait until the TrayAllocator releases the induction waiting for an empty tray. Finally the induction controller will wait a time equal the induction rate multiplied by tray steps, in order to keep the separation between items.

```

thread
(
    while (ItemsToInduct()) do
    (
        let item = GetFirstItem() in
            allocator.RequestTray(threadid, selfIC, item);
        allocator.Wait();
        SorterEnviroment`trayStep.WaitSteps(InductionRate);
    );
);

```

3.2.5. The tray allocator class

The tray allocator class is responsible for associating an empty tray to each item that is present at the inductions. The tray allocator is associated with every induction controller that is present at the system. In this particular case, it will be interacting with three different running induction controller threads. The concurrency is a major issue that should be considered in this class.

The tray allocator is keeping track on the inductions and items by using a map. The domain is composed by a natural number (the induction controller thread id) and the range is composed by a product type, formed by the Induction Controller and the correspondent item.

```

itemsToInduct : map nat to (InductionController * Item) := {|->};

```


The synchronization in this class is done as follows. There is a mutex on the RequestTray operation, so just one induction is allowed to request a tray at time. It is necessary with a mutex to ensure synchronization between the Induction Controller and the Tray allocator threads, so the operations RequestTray and CheckItemsToInduct cannot be invoked simultaneously. A permission predicate is used to ensure that the wait operation is executed only when the thread id is not in the set composed by the domain itemsToInduct. That means the inductionController is not already contained in the map itemsToInduct. These synchronization constraints are modelled as follows:

```
sync
    mutex(RequestTray);
    mutex(RequestTray, CheckItemsToInduct);
    per Wait => threadid not in set dom itemsToInduct;
```

The thread part of the class is explained in the following lines. The tray allocator is executing continuously three actions. First of all, it is waiting a time equal to one tray step. The CardReader operation is invoked to simulate that the sorter ring has moved one tray step. Finally the CheckItemsToInduct operation is invoked. The items contained at the waiting inductions will be inducted. Further details on this operation will be provided in the lines below.

```
thread
    while (true) do
    (
        SorterEnviroment`trayStep.WaitSteps(1);
        CardReader(TrayStep`GetCounts() mod
                    TrayAllocator`NumOfTrays + 1, <Empty>);
        CheckItemsToInduct();
    );
```

The CheckItemsToInduct operation will induct the items for all the waiting inductions on the sorter. The operation is using the “for all” quantifier to go through all the induction controllers waiting. A record composed by the induction controller and the associated item is created out of the itemsToInduct map. The operation tries to induct the first item contained at the inductions and it releases the induction that is waiting with the item to induct. In the case it was not possible to induct the item the priority associated to the induction is incremented. In the sequential model the induction controllers was maintaining the priority. In this way, it will be the tray allocator to decide on the order to induct item in the next iteration of a tray step.

```

CheckItemsToInduct: () ==> ()
CheckItemsToInduct () ==
(
  for all t in set dom itemsToInduct
  do
    let mk_(ic, item) = itemsToInduct(t)
    in
      if InductItem(ic, item) then
        (
          ic.InductFirstItem();
          ReleaseWaitingIC(t);
        )
      else
        ic.IncrementPriority();
);

```

The RequestTray operation is called from the induction controller thread in order to get an empty tray. In this way, the induction will be able to induct an item. The operation is mainly calling a second operation called AddItem, which is explained below. A precondition is used in order to state that the threadid associated to the induction controller is not already present at the itemsToInduct map.

```

Public RequestTray: nat * InductionController * Item ==> ()
RequestTray (t, ic, item) ==
  AddItem(t, ic, item)
pre t not in set dom itemsToInduct;

```

The AddItem operation is adding the induction controller thread id to the itemsToInduct map. This is done by using the map union expression by adding a single element map composed by the threadID and a tuple created with a reference to the induction controller and item. Same precondition as in the above RequestTray operation.

```

AddItem: nat * InductionController * Item ==> ()
AddItem (t, ic, item) ==
  itemsToInduct := itemsToInduct munion {t |->mk_(ic, item)}
pre t not in set dom itemsToInduct;

```

The `releaseWaitingIC` operation is removing the waiting induction threads that are waiting with an item to induct. This is done by the operator restricted by. Same precondition as in the above `RequestTray` operation.

```
-- Remove induction waiting with item to induct
ReleaseWaitingIC: nat ==> ()
    ReleaseWaitingIC (t) ==
        itemsToInduct := {t} <-: itemsToInduct
pre t in set dom itemsToInduct;
```

A simple wait operation is used to block the thread until it is removed from the waitingICs waiting map.

```
public Wait: () ==> ()
    Wait() == skip;
```

4. Real-time Distributed model

In this chapter the Real Time and distributed model of the tray allocator will be presented from both UML and VDM++ perspective.

The real time version is all about time. The RT VDM++ scheduler executes the model with its own time ticks which we will synchronize with the time in ms feeding items by the sorter environment. We will introduce a periodic thread in the Tray allocator that simulates the tray step. The induction controller will also have a periodic thread that ensures the separation between items according to the induction rate.

The synchronization between the real time ticks and the time measured in milliseconds is calculated by using the following expression:

$$TimeUnits[time\ units\ per\ ms] = \frac{TrayStepTimeUnits}{\frac{(TraySize \times 1000)}{Speed}}$$

Example:

TrayStepTimeUnits = 6000 [time units] (Periodic thread in Tray allocator)

TraySize = 600 [mm]

Speed = 2000 [mm/sec]

TimeUnits = 6000 / ((600 x 1000)/2000) = 20 [time units per ms]

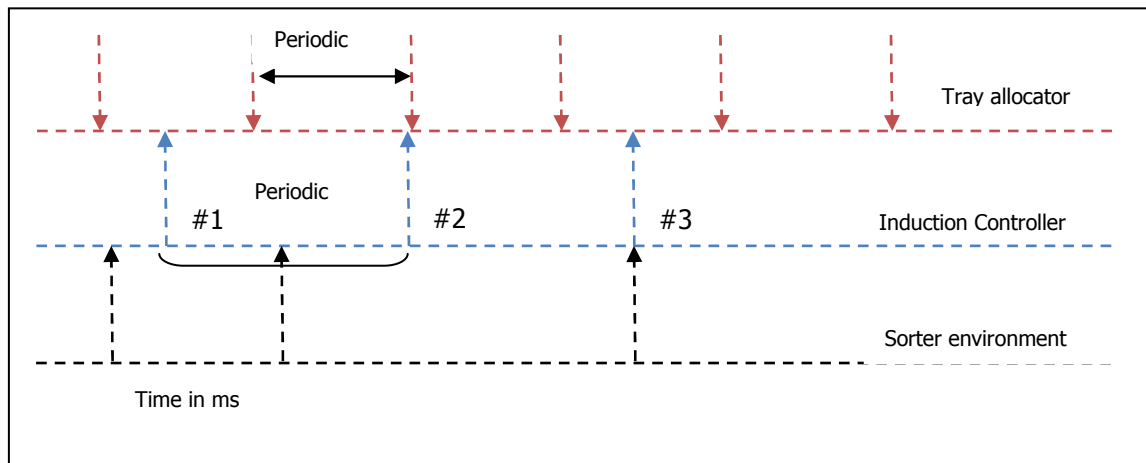


Figure 6: ms – simulation time equivalence

4.1. System deployment

The Sorter Controller class is changed to be the “system” that will create the whole system and deploy objects to CPU’s. In this sense it has to create the different objects, CPU’s and buses the system is going to require. In the following model snippet it can be seen how four CPU’s are created, with the scheduling policy FCFS (First Come First Served) and with a virtual frequency of 1E6.

```

system SC
    instance variables
        cpuIC1 : CPU := new CPU (<FCFS>,1E6);
        cpuIC2 : CPU := new CPU (<FCFS>,1E6);
        cpuIC3 : CPU := new CPU (<FCFS>,1E6);
        cpuIC4 : CPU := new CPU (<FCFS>,1E6);
        cpuTA : CPU := new CPU (<FCFS>,1E6);

```

The different objects that contain threads to be deployed on different CPUs should be instantiated in the system. Four different instances of the InductionController are created and grouped in a sequence called inductionGroup. A tray allocator object is created as well, receiving as a parameter the sequence of induction controllers that have been declared in the previous line.

```

public static ic1 : InductionController := new InductionController(1);
public static ic2 : InductionController := new InductionController(2);
public static ic3 : InductionController := new InductionController(3);
public static ic4 : InductionController := new InductionController(4);
public static inductionGroup : seq of InductionController
                                := [ic1, ic2, ic3, ic4];
public static allocator : TrayAllocator := new
TrayAllocator(inductionGroup);

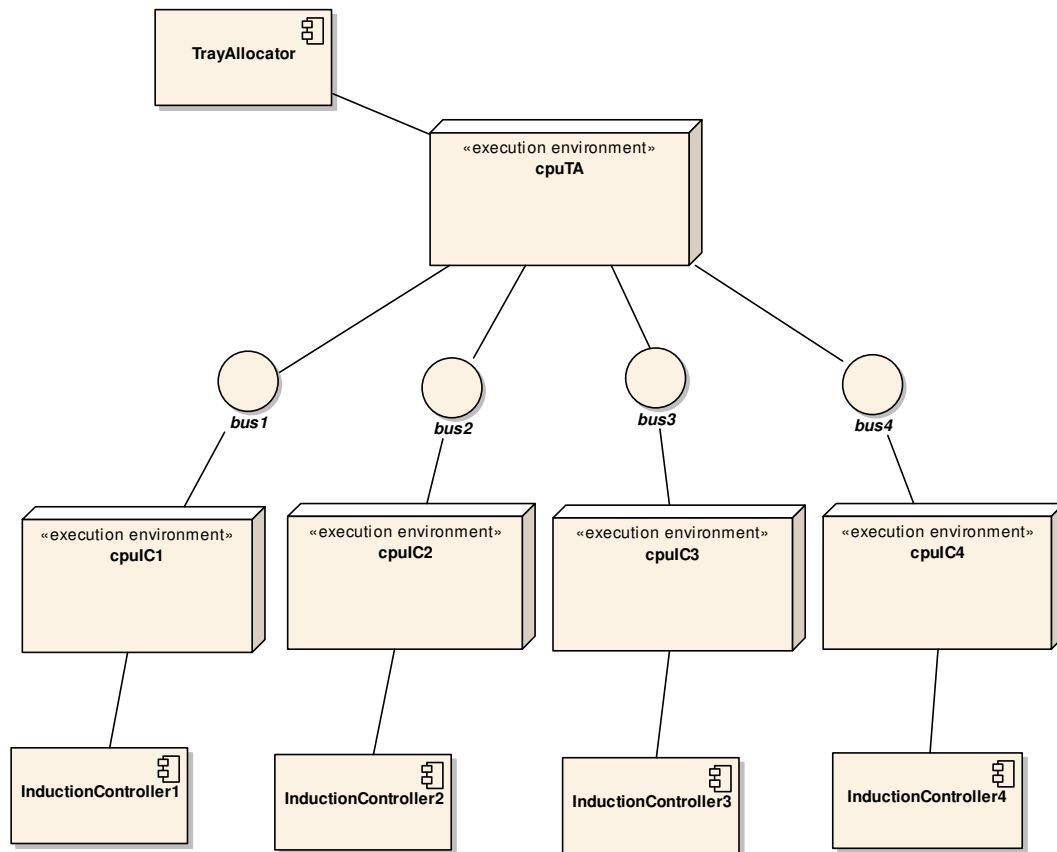
```

Once CPU's and objects have been created depending on the deployment scenario under study they are going to be connected through the busses and CPU's. In the following sections two different deployments will be explained.

4.1.1. Scenario 1

In the first scenario five different CPU's are used to deploy the different induction controllers and the tray allocator. The cpuTA which will be running the TrayAllocator will have a point to point connection through buses 1, 2, 3 and 4 with each induction controller, running on cpuIC 1, 2, 3 and 4.

From the UML perspective, this deployment is illustrated in figure 6.

**Figure 7: Deployment scenario 1**

In this deployment four busses are used to establish a point to point communication between the CPUs running the induction controllers and the CPU running the tray allocator.

```
bus1 : BUS := new BUS (<FCFS>, 1E3, {cpuIC1, cpuTA});
bus2 : BUS := new BUS (<FCFS>, 1E3, {cpuIC2, cpuTA});
bus3 : BUS := new BUS (<FCFS>, 1E3, {cpuIC3, cpuTA});
bus4 : BUS := new BUS (<FCFS>, 1E3, {cpuIC4, cpuTA});
```

After components have been deployed and busses have been defined, it is possible to deploy the whole system.

```
public SC: () ==> SC
SC() ==
(
    cpuIC1.deploy(ic1);
    cpuIC2.deploy(ic2);
    cpuIC3.deploy(ic3);
    cpuIC4.deploy(ic4);
    cpuTA.deploy(allocator);
);
```

4.1.2. Scenario 2

In the second scenario three different CPU's are used to deploy the different induction controllers and the tray allocator. The `cpuTA` which will be running the `TrayAllocator` will have a point to point connection through buses 1 and 2 with two induction controllers running on `cpuIC 1` and `2`. We don't expect there should be any difference between deployment scenario 1 and 2 since all communication is done between the `TrayAllocator` and `InductionControllers`. If we did have communication traffic between the `InductionControllers` we would expect to see a difference.

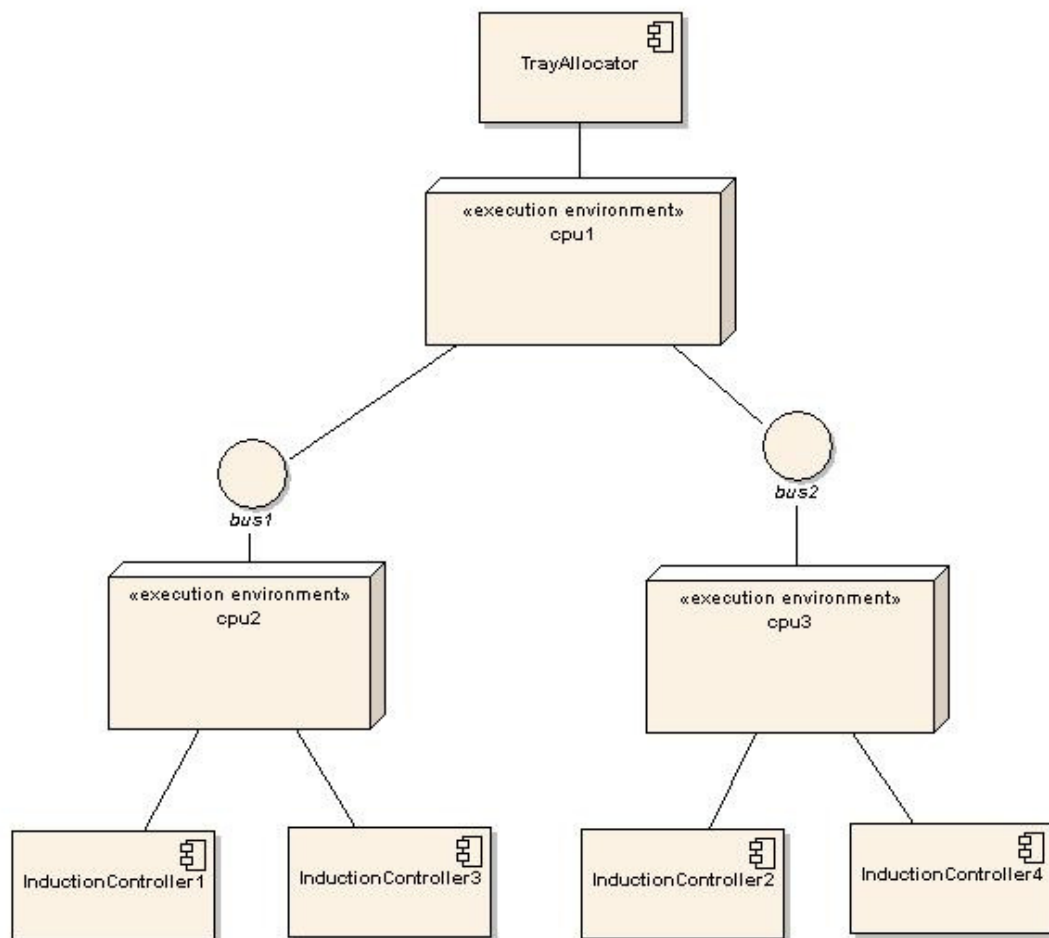


Figure 8: Deployment scenario 2

```

cpuIC1.deploy(ic1);
cpuIC1.deploy(ic2);
cpuIC2.deploy(ic3);
cpuIC2.deploy(ic4);

```

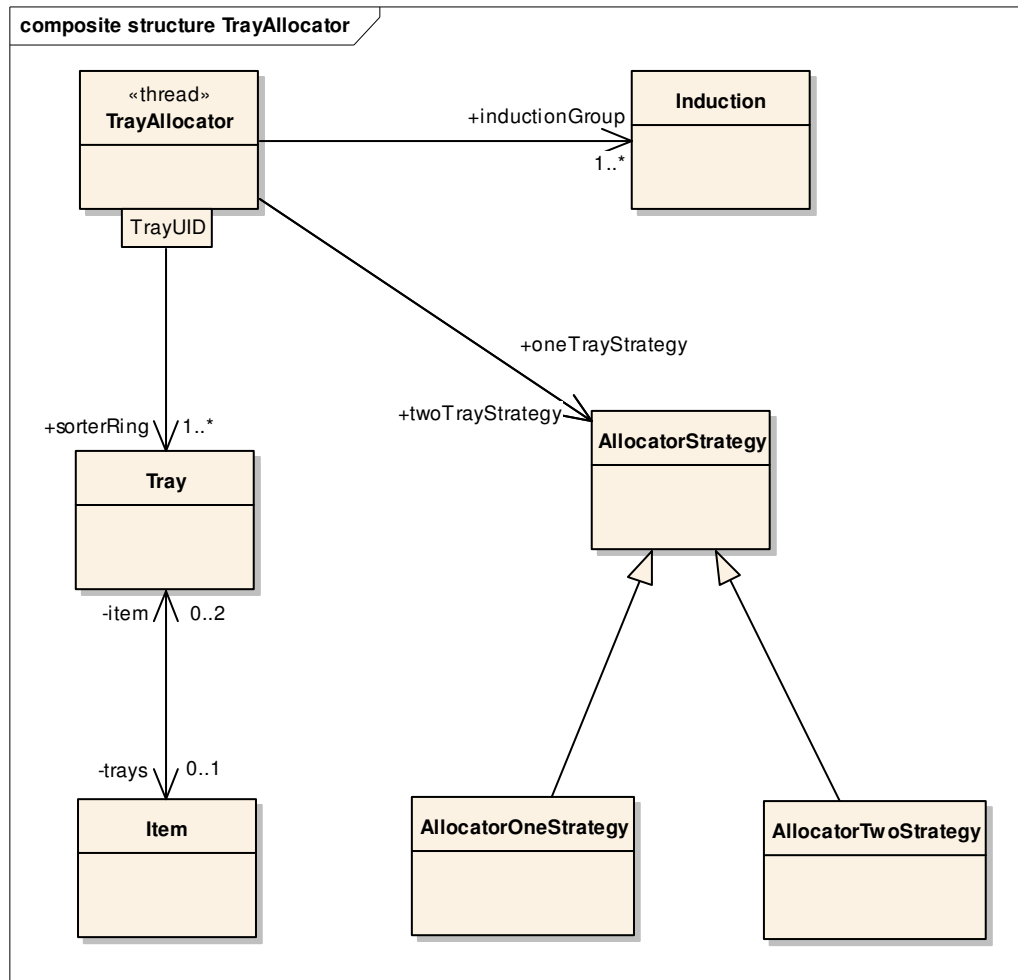


Figure 9 UML Composite structure for TrayAllocator deployed on cpuTA

As shown in the UML composite diagram above there are a number of objects that must be instantiated on the TrayAllocator CPU (cpuTA). The real time distributed model must be modified to ensure that objects are created and running on this CPU to minimize bus traffic between the induction controllers and the tray allocator. We have introduced the new class **Induction** that models the functionality of the IC required by the TrayAllocator to minimize traffic between the cpuTA and cpuIC. On the InductionController CPU (cpuIC#) we only have to create objects of the **Item** and **InductionController** classes as illustrated below.

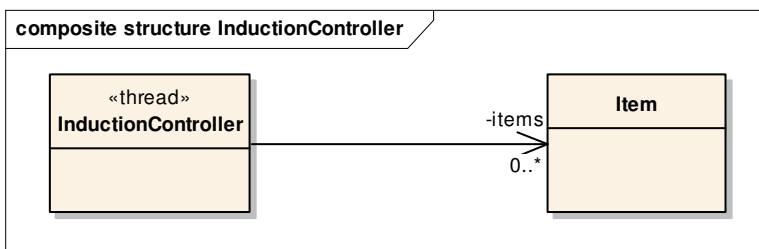


Figure 10 UML Composite structure for InductionController deployed on cpuIC#

4.2. The real time model in VDM++*4.2.1. The SorterEnvironment class*

The SorterEnvironment class is responsible for starting the simulation and all the active threads present in the system. It is responsible as well for checking if the simulation has finished or not and feeding items to the system through the associated ItemLoader class. Here the same setup is used as for the concurrent model.

Prior to simulation start, the different relations between the inductions and the tray allocator class should be done. This is performed by the CreateAssignments operation. The operation is accessing the static variables at the sorter controller that represent the induction controllers, and invoking the AssignAllocator operation, with the allocator instance as a parameter. Finally the induction group that is composed by the already instantiated inductions is assigned.

```
CreateAssignments: () ==> ()
CreateAssignments () ==
(
    SC`allocator.CreateAllocatorObjs();
    SC`ic1.AssignAllocator(SC`allocator);
    SC`ic2.AssignAllocator(SC`allocator);
    SC`ic3.AssignAllocator(SC`allocator);
    SC`ic4.AssignAllocator(SC`allocator);
    AssignInductionGroup(SC`inductionGroup);
);
```

Since the SorterEnvironment class is responsible for initiating and finishing the whole simulation as we did have in the concurrent model.

```
sync
    per isFinished => not busy;
```

The actions that are performed while the sorter environment thread is executed are described in the following lines. First of all, the CreateAssignment operation is invoked, so the induction controllers and the tray allocator are properly associated since now this is not possible to create in the SC system class where all objects are created statically.

```
thread
(
    CreateAssignments();
```

After the associations have been created, all the threads in the system are started. This is done in the same way as for the concurrent model.

```
start (SC`allocator);
for all i in set {1,...,TrayAllocator`NumOfInductions}
    do start (inductionGroup(i));
```

We have defined a constant TUinMS that is computed as how many simulation time ticks correspond to a millisecond. This calculation is based on the periodic thread in the TrayAllocator that is executed after a number of time tick units defined by the TrayStepTimeUnits.

While the simulation is running the sorter environment is triggering the following actions in the case the time has gone beyond the scheduled time in nextMs interval. The algorithm goes through all the inductions modeled in the system and gets the correspondent item for that induction at time specified in ms where it is supposed being fed to the induction. The items are provided by the itemLoader class. If an item is going to be fed, the itemID is incremented and the item is fed to the induction contained at the induction group. The item is now created by the InductionController instead of the environment. Finally the millisecond counter is incremented with the StepMS value. In the below model we simulate feeding items with a resolution of 100 ms.

```

public StepMs : nat = 100;
public TUinMS : nat = (StepMs * TrayAllocator`TrayStepTimeUnits)
                        / ((Tray`Size * 1000) / Speed);

while busy do
(
    if (time > nextMs)
    then
    (
        nextMs := time + TUinMS;
        for all i in set {1,...,TrayAllocator`NumOfInductions}
        do
        (
            -- Check for item to feed induction at time step
            let size = itemLoader.GetItemAtTimeStep(msCount, i)
            in
                if (size > 0)
                then
                (
                    itemId := itemId + 1;
                    inductionGroup(i).FeedItem(size, itemId);
                );
            msCount := msCount + StepMs;
        );
    );
);

```

Inside the same while busy loop and just after the previous operations, the condition for stop of the simulation is checked. In the case the elapsed simulation time has gone beyond the value specified in the simulation scenario, the simulation is stopped by calling an operation in the allocator to prevent the periodic thread in this class to stop checking items to be inducted.

```

-- Check if simulation is finish
if (time >= itemLoader.GetNumTimeSteps() * (TUinMS/StepMs)) then
(
    SC`allocator.StopSimulation();
    busy := false;
);

```

4.2.2. The InductionController class

The induction controller class is responsible for modelling each induction. The thread of this class are communicating with the tray allocator thread. Each induction controller can receive an

item inducted from the environment. Since this can happen at any time a `FeedItem` asynchronous operation has been added to our model.

The `FeedItem` operation is receiving the item id and the size. Compared with the concurrent model a new `Item` is now created here. This is important since now items has to be created on the induction controller CPU to minimize communication overhead over the busses. This operation is illustrated in the following model snippet.

```

async
public FeedItem: nat * nat ==> ()
FeedItem(ucid, size) ==
(
    items := items ^ [new Item(ucid, size)];
);

```

The `GetFirstItem` operation provides a small yet vital operation that returns the first item present at the induction. This operation is applying the head operator over the sequence of items declared as an instance variable at the class. A precondition is used to state that the length of this sequence should be different that 0, that means the sequence is not empty and items are present at the induction.

```

GetFirstItem: () ==> Item
GetFirstItem() ==
    return hd items
pre len items <> 0;

```

The Induction Controller provides a function to induct an item present at the induction in to the sorter. This function is the `InductFirstItem`, which is applying the tail operator over the sequence of items and assigning the result to it. Again this operation should be protected by a precondition stating that the sequence of items should be different from 0, meaning that there are items at the induction.

```

public InductFirstItem: () ==> ()
InductFirstItem() ==
    items := tl items
pre len items <> 0;

```

The `ItemsToInduct` functions provide a simple mechanism to check if items are contained at the induction or not.

```

ItemsToInduct: () ==> bool
ItemsToInduct () ==
    return len items <> 0;

```

A wait function is provided so the induction controller thread can wait until it is removed from the waiting map `waitingICs`. This function is following the same structure like the one introduced in previous chapter but now moved from the `TrayAllocator` to the `InductionController`. This is important as we will see later for the permission predicate on the `Wait` operation.

```
Wait: () ==> ()
      Wait() == skip;
```

The asynchronous operation WaitInductItem is selecting the first item present at the induction and requesting an empty tray at the tray allocator CPU. The call will block and wait until a tray has been located. Once the step is completed the first item will be inducted by executing the operation InductFirstItem.

```
async
  WaitInductItem: () ==> ()
  WaitInductItem() ==
    let item = GetFirstItem()
    in
      (
        allocator.RequestTray(threadid, id, item);
        Wait();
        InductFirstItem();
      );
```

An operation called InductStep is responsible for checking if there are items present at the induction. In the case there are, the asynchronous operation WaitInductItem previously presented will be invoked.

```
InductStep: () ==> ()
InductStep() ==
  if (ItemsToInduct()) then
    WaitInductItem();
```

The synchronization mechanisms applied for the induction controller class are defined by the means of a number of mutexes and permission predicates. Both elements are presented in the following model snippet. A mutex for the FeedItem and WaitInductItem is specified to ensure they are not be invoked simultaneously from different threads. A third mutex is used for the FeedItem and the InductFirstItem operations, because they are called from the environment and induction controller threads.

```
sync
  mutex (FeedItem);
  mutex (FeedItem, InductFirstItem);
  mutex (WaitInductItem);
per Wait => threadid not in set dom allocator.icThreadsWaiting;
```

The InductStep is running as a periodic thread being constantly executed with a prefixed frequency of 12000 time units equal to the induction separation. The induction separation is equal to 2*tray step time ticks, where a tray step time tick is specified by the periodic thread TrayStep on the TrayAllocator CPU.

```
thread
  periodic (12000, 0, 0, 0) (InductStep);
```

4.2.3. The induction class

In order to improve the performance of the system, a new class called Induction has been introduced in to the model. This class will be associated to the Tray Allocator and it will contain information concerning the induction id and the current priority. This is due to the fact that the induction controllers and the tray allocator are deployed in different cpu's, so whenever the tray allocator needs to get information about the induction it does not need to communicate over the bus.

```
instance variables
  priority : nat := 0;
  id : nat1;
```

4.2.4. The tray allocator class

The tray allocator class is responsible for allocating an empty tray to items that should be inducted into the sorter. This class is communicating with the induction controllers threads. Since the operations and thread in this class is communicating with a number of concurrent objects, some considerations should be taken. The class is using a mutex over the RequestTray operation, so two different induction controllers cannot request a tray at the same time. A second mutex is used in order to prevent simultaneous execution of RequestTray and CheckItemsToInduct. This ensures synchronization between the InductionController and the TrayAllocator class.

```
sync

mutex(RequestTray);
mutex(RequestTray, CheckItemsToInduct);
```

The tray allocator is running a periodic thread every 6000 time tick units, and executing periodically the TrayStep operation. The 6000 time tick units are selected by measurement for the worse case execution time of the TrayStep operation.

```
thread
  -- Time units to simulate a tray step
  periodic (6000, 0, 0, 0) (TrayStep);
```

The TrayStep operation that is referred in the previous paragraph is explained below. Its main purpose is to simulate a tray step in the system. In case the simulation is not stopped by the environment the busy flag will be true. The tray counter value is incremented by one (one tray step). The CardReader call simulates that the sorter has moved one tray step and, since the sorter is a one surfaces continuous ring, this phenomena is modelled with a modular operation. Finally the items for all waiting induction threads are inducted.

```
TrayStep: () ==> ()
TrayStep () ==
(
  if (busy) then
  (
    trayCount := trayCount + 1;
```

```

    CardReader(trayCount mod TrayAllocator`NumOfTrays + 1);
    CheckItemsToInduct();
);

```

The operation `CheckItemsToInduct` is looking for waiting items to induct by looking for thread id's in the domain of the map `itemsToInductMap`. The domain is defined by thread id's and the range by a product composed by references to the Induction and Item objects. That is how we determine if there are items to induct at the inductions. The operation is going through the thread id's defined by the domain and inducting each registered item. In the case that the item could be inducted successfully by the `InductItem` operation, the priority associated to the induction is cleared (`ic.ClearPriority`) and the induction controller that was block is released (`ReleaseWaitingIC`). If it has not been possible to induct the item on to the sorter, the associated priority to the induction is incremented. Since the new induction class is deployed at the same CPU as the `TrayAllocator` the operations `ClearPriority` and `IncrementPriority` will not cause and traffic on the bus between the `cpuIC#` and `cpuTA`.

```

CheckItemsToInduct: () ==> ()
CheckItemsToInduct () ==
(
    for all t in set dom itemsToInductMap
    do
        let mk_(ic, item) = itemsToInductMap(t)
        in
            if InductItem(ic, item) then
                (
                    ic.ClearPriority();
                    ReleaseWaitingIC(t);
                )
            else
                ic.IncrementPriority();
);

```

It is possible from the induction controller threads to invoke the `RequestTray` operation in order to notify the tray allocator that an item has to be inducted. The `RequestTray` operation is receiving as parameters the thread id corresponding to the induction controller, the induction id and finally a reference to the item. The reference to the induction is obtained from the induction group and it is passed as a parameter, together with the treadid and the reference, to induction to the `AddItem` operation. See later in chapter 5.2 how we have tried to optimize this operation by creating a new instance of the item on the `TrayAllocator` CPU.

```

public RequestTray: nat * nat * Item ==> ()
RequestTray (tid, icid, item) ==
    let ic = inductionGroup(icid)
    in
        AddItem(tid, ic, item)
pre icid in set inds inductionGroup and tid not in set dom
itemsToInductMap;

```

The AddItem operation introduced previously is described in the following lines. This operation adds a maplet to the itemsToInductMap by using the map union operation. The maplet is composed by the thread id (domain) and a record storing induction controller and item (range). A similar operation is done with the map icThreadsWaiting. The maplet added in this case is composed by the thread id and the induction id. A precondition is used to express that the thread id should not be already contained in the itemsToInductMap.

```
-- Add induction waiting with item to induct
AddItem: nat * Induction * Item ==> ()
AddItem (t, ic, item) ==
atomic (
  itemsToInductMap := itemsToInductMap munion {t |-> mk_(ic, item)};
  icThreadsWaiting := icThreadsWaiting munion {t |-> ic.GetId()}
)
pre t not in set dom itemsToInductMap;
```

The operation ReleaseWaitingIC is responsible to remove the waiting thread id from both itemsToInductMap and icThreadsWaiting maps. This is done by using the restricted by map operator. As a result of this, the induction is released (unblock) and its thread execution can continue.

```
ReleaseWaitingIC: nat ==> ()
ReleaseWaitingIC (t) ==
atomic (
  itemsToInductMap := {t} <-: itemsToInductMap;
  icThreadsWaiting := {t} <-: icThreadsWaiting
)
pre t in set dom itemsToInductMap;
```

```
public TrayAllocator: () ==> TrayAllocator
TrayAllocator() ==
(
  -- CreateAllocatorObjs();
);
```

The CreateAllocatorObjs is used in order to instantiate all the necessary objects for the thread allocator execution. If this operation is called from the constructor all objects will be located on the vCPU, therefore the call is moved to the SorterEnvironment where it is called just before starting the TrayAllocator thread. This is shown in the following lines.

```
public CreateAllocatorObjs: () ==> ()
CreateAllocatorObjs() ==
(
  sorterRing := {num |-> new Tray(num) |
    num in set {1,...,NumOfTrays}};
  inductionGroup := [new Induction(id) |
    id in set {1,...,NumOfInductions}];
  oneTrayStrategy := new AllocatorOneTray(self);
  twoTrayStrategy := new AllocatorTwoTray(self);
);
```

5. Test setup

This chapter introduces how the concurrent and real-time tray allocation model has been tested by using proof obligations, interactive debugger and stimuli files, that contain different simulation scenarios with several item arrival distributions.

The system test has been performed by reading test files that contain a scenario of items to feed the inductions during simulation of time instead of tray steps that was used in the sequential model. A simulation scenario file contains a sequence of tuples for every item that must be created during simulation. A tuple is constructed by the time in milliseconds, induction id and item size. In the example below the simulation time steps is specified to 4 feeding items at time step 1 and 3. At time step 1 inductions 1, 2, 3 are fed with items of size 100, 200 and 900 mm. The scenario files from the sequential model are changed by multiply of tray steps with 300. This is the time in ms it takes a tray of size 600 mm to move one step at a sorter speed of 2.0 m/s. ($0.6[m]/2.0[m/s]*1000[ms]$)

Example for contents of scenario file:

```
mk_(1200, [mk_(300,1,100), - total simulation time 1200 ms
          mk_(300,2,200), - time 300 ms, induction id 2, item size 200
          mk_(300,3,900),
          mk_(900,1,700), - time 900 ms, induction id 1, item size 700
          mk_(900,2,600)])
```

5.1. Concurrent version setup

As described in chapter 3. The concurrent version is extended with the TimeStamp and TrayStep classes that are used to simulate time in milliseconds. The TrayStep class is used to synchronize the time in ms it takes to move a tray one step at the speed of the sorter.

The ItemLoader class is the same as for the sequential model and it handles the reading of the scenario files. The ItemLoader now uses the time in ms when calls are performed to the (GetNumTimeSteps) and (GetItemAtTimeStep). The ItemLoader is created by the World class and the SorterEnvironment which uses the ItemLoader during simulation to feed items to the inductions. The TimeStamp class is used by the SorterEnvironment to synchronize the simulation time with the contents of the scenario file. The environment feeds new items to the induction specified by the simulation time incremented by the environment.

See UML class diagram below of how the new classes is associated to the World and SorterEnvironment.

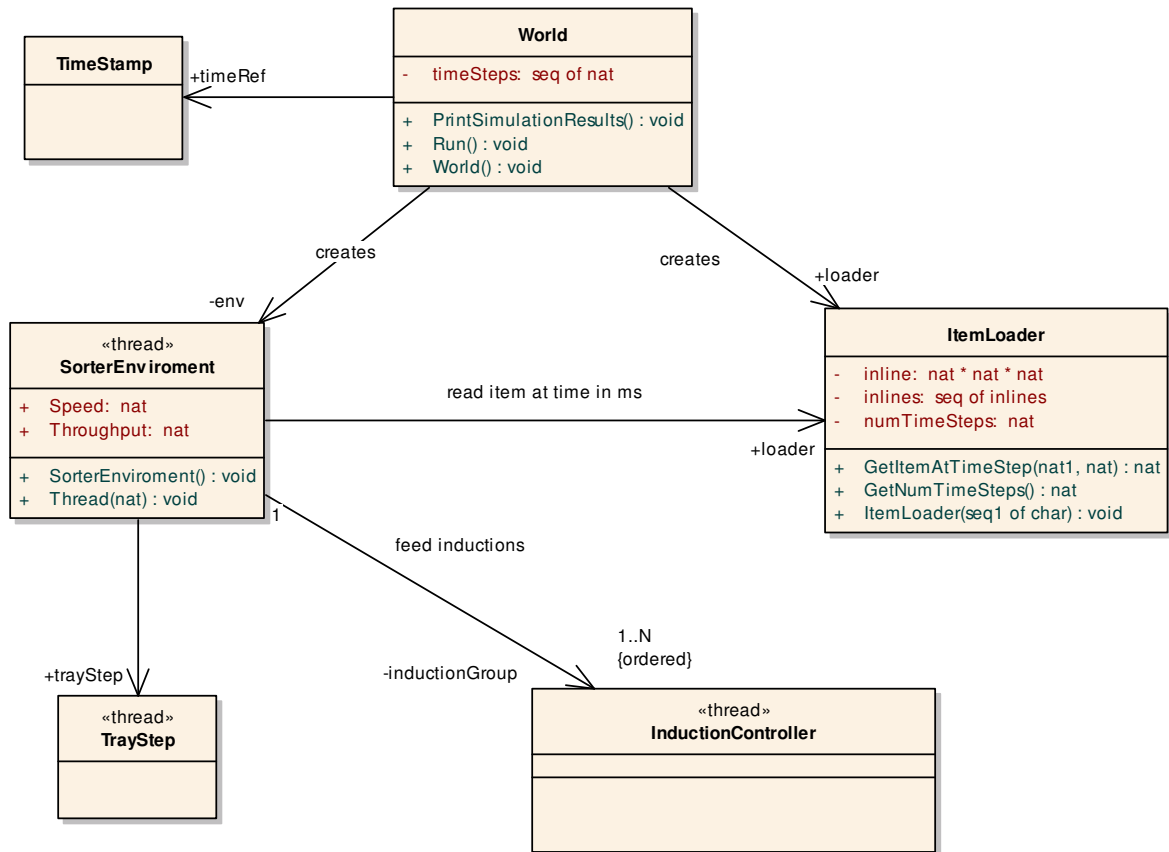


Figure 11: Class diagram concurrent model including the TimeStamp and TrayStep classes

When the simulation of time in ms is completed a summary report is printed with the current configuration of the model and the key important figures of the simulation results. Example of simulation summary report created by the operation `PrintSimulationResults()` in the world class:

```

-----
Tray allocation Conc model #1: scenario3.txt
- Mix of one and two tray items
-----
< 1 >< 2 >
*Induction id 3
-> Item id 3 size 200 on tray id 10
*Induction id 2
-> Item id 2 size 800 on tray id 7
-> Item id 2 size 800 on tray id 8
*Induction id 1
-> Item id 1 size 100 on tray id 6
< 3 >< 4 >< 5 >
*Induction id 2
-> Item id 5 size 400 on tray id 11
*Induction id 1
-> Item id 4 size 200 on tray id 9
< 6 >
*Induction id 3
-> Item id 6 size 700 on tray id 13
-> Item id 6 size 700 on tray id 14
< 7 >< 8 >< 9 >
  
```

```

*Induction id 2
-> Item id 8 size 300 on tray id 15
.....
< 15 >< 16 >< 17 >< 18 >< 19 >< 20 >
*Induction id 1
-> Item id 10 size 900 on tray id 3
-> Item id 10 size 900 on tray id 4
< 21 >< 22 >< 23 >< 24 >
*Induction id 3
-> Item id 12 size 200 on tray id 12
-----
Simulation completed for sorter configuration
-----
< 25 >Specified throughput [items/hour]: 10000
Sorter speed                [mm/sec]: 2000
Item max size                [mm]: 1500
Item min size                [mm]: 100
Tray size                    [mm]: 600
Number of trays              : 20
Number of inductions         : 3
Induction rate               : 2
Induction separation         [trays]: 2
-----
Number of trays with items    : 17
Two tray items on sorter     : 5
Number of tray steps         : 24
Number of inducted items     : 12
Calculated throughput[items/hour]: 8500
-----
      ****  Sorter is not full  ****
-----

```

The tray numbers in < > is printed by the TrayStep thread to indicate the current tray at the card reader.

In the example above the calculated throughput is below the specified throughput. We have in this example more empty trays (Number of trays 20 – Number of trays with items 17). The simulation contains a mix of one and two tray items feeding the inductions during simulation. We see that the order the inductions is inducing items on the sorter is not the same every time. In the sequential model we always serviced inductions in the same order 1, 2, 3. Since now the inductions and tray allocator is running in separated threads they are not serviced in any predictable order. If we look at the scenario test for two tray items, we see now that empty trays are created on the sorter and we cannot fill the sorter as we could for the sequential model. In the scenario below we are missing tray id 15 due to different order the inductions are serviced.

```

-----
Tray allocation Conc model #1: scenario2.txt
- Only two tray items
-----
< 1 >< 2 >
*Induction id 3
-> Item id 3 size 800 on tray id 9
-> Item id 3 size 800 on tray id 10

```

```

*Induction id 2
-> Item id 2 size 700 on tray id 7
-> Item id 2 size 700 on tray id 8
*Induction id 1
-> Item id 1 size 600 on tray id 6
-> Item id 1 size 600 on tray id 5
< 3 >< 4 >< 5 >< 6 >
*Induction id 2
-> Item id 5 size 600 on tray id 11
-> Item id 5 size 600 on tray id 12
< 7 >< 8 >< 9 >< 10 >
*Induction id 1
-> Item id 4 size 900 on tray id 13
-> Item id 4 size 900 on tray id 14
< 11 >*Induction id 3
-> Item id 6 size 700 on tray id 18
-> Item id 6 size 700 on tray id 19

```

3 test scenarios (Appendix C) are executed for one tray items, two tray items and a mix of one and two tray items. The algorithm is able to fill the sorter without starvation in case of only one tray items. The concurrent model leaves empty trays on the sorter when we have two tray items only, or a mixture of one and two tray items. Its performance is especially poor with 2 tray items when we compare with the sequential model.

Concurrent model results:

- | | |
|------------------------------|--|
| 1. One tray items only | -> Calculated throughput[items/hour]: 10434 (full) |
| 2. Two tray items only | -> Calculated throughput[items/hour]: 8347 (4 empty) |
| 3. Mix of one/two tray items | -> Calculated throughput[items/hour]: 8500 (3 empty) |

Sequential model results:

- | | |
|------------------------------|--|
| 1. One tray items only | -> Calculated throughput[items/hour]: 10909 (full) |
| 2. Two tray items only | -> Calculated throughput[items/hour]: 10909 (full) |
| 3. Mix of one/two tray items | -> Calculated throughput[items/hour]: 9272 (3 empty) |

During the testing we have found and fixed a number of errors in the model especially it has this time been hard to get the model correct in synchronization between time and trays step and ensuring correct mutex and permission predicates.

5.2. Real time version setup

The simulation setup by reading scenario files that specifies the time in ms when items to be inducted on different the inductions is the same as for the concurrent model. The simulation setup is done for two different deployments as described in chapter 4.1 for scenario 1 and scenario 2.

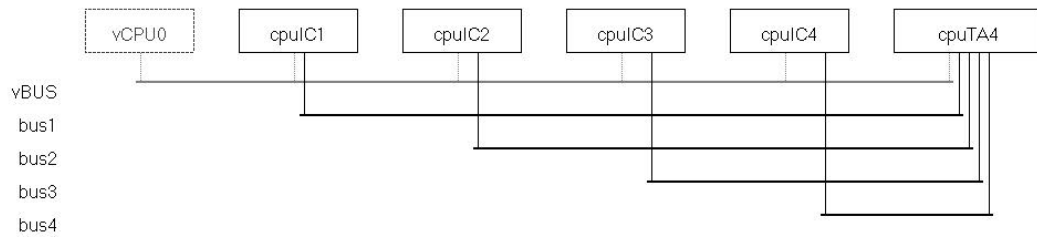


Figure 12 Deployment scenario 1 (CPU for each IC)

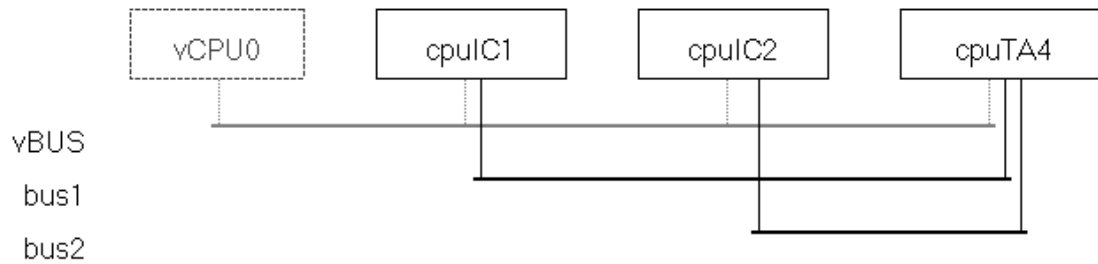


Figure 13 Deployment scenario 2 (CPU for two ICs)

A new scenario file has been made to test each deployment scenario that contains a mixture of one and two tray items to be inducted on the four different inductions.

During our first test of the real time version, we did find a number of problems with our initial RT model, that we have not described in this document. This first model was the concurrent model deployed to CPU's for each induction controller and a CPU for the central TrayAllocator. This model was only modified to handle the synchronization between time in ms, simulation time ticks and tray steps without thinking about object deployment. This first model we managed to get working, but the simulation was extreme slow. The periodic thread on the Tray Allocator was using ~20000 time ticks to execute. After being able to analyse the log files we discovered that there was a heavy interaction between the InductionController and TrayAllocator over the buses and between CPUs. Objects like items and trays was running on the virtual CPU and not as where we expected. Seen part of log file below where we have a lot of readings from the TrayAllocator to tray objects located on the virtual CPU.

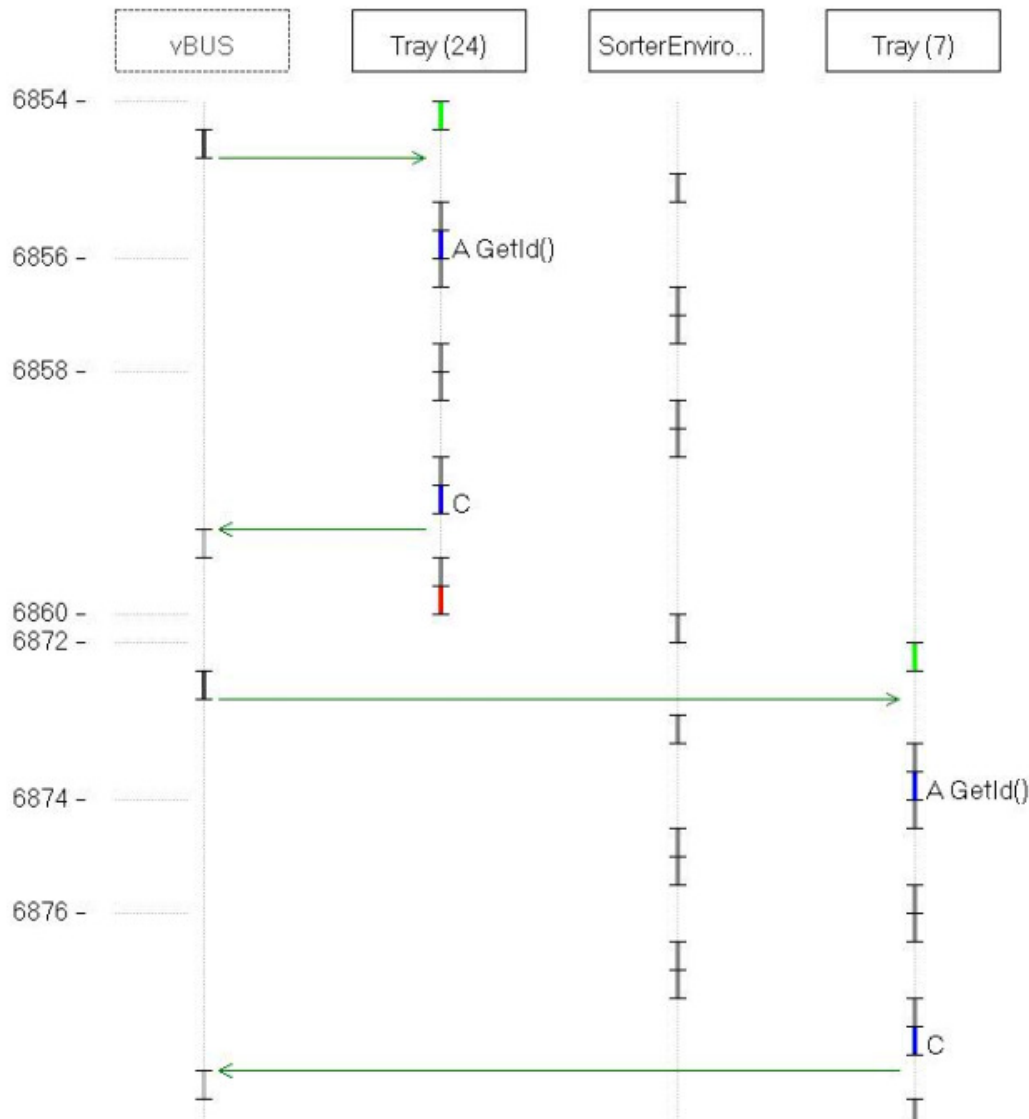


Figure 14 View part of simulation log for first non optimized RT model

Therefore we decided to change the model as described in the chapter 0. We minimized the interaction between the TrayAllocator and InductionController by introduction of the Induction class that contained the priority and IC id being part of the TrayAllocator deployed to cpuTA. The only interaction between TrayAllocator and InductionController was reduced to a call to the operation `RequestTray` on the TrayAllocator. We removed the priority from the InductionController and used the IC id to synchronize the TrayAllocator and InductionController.

In this first version of the model the thread in the InductionController was not made periodic. We made the IC thread waiting by calling an operation `BusyWait` that made a loop reading the time (Time slices) to simulate the separation between inducing items see log in figure 13. We discovered that this method was not working due to the way the VDM-RT scheduler is advancing time based on the smallest time step calculated over all resources waiting to run. So

if a busy loop is made or the duration statement is used nothing else will be running on this CPU before the task is blocked waiting on a resource.

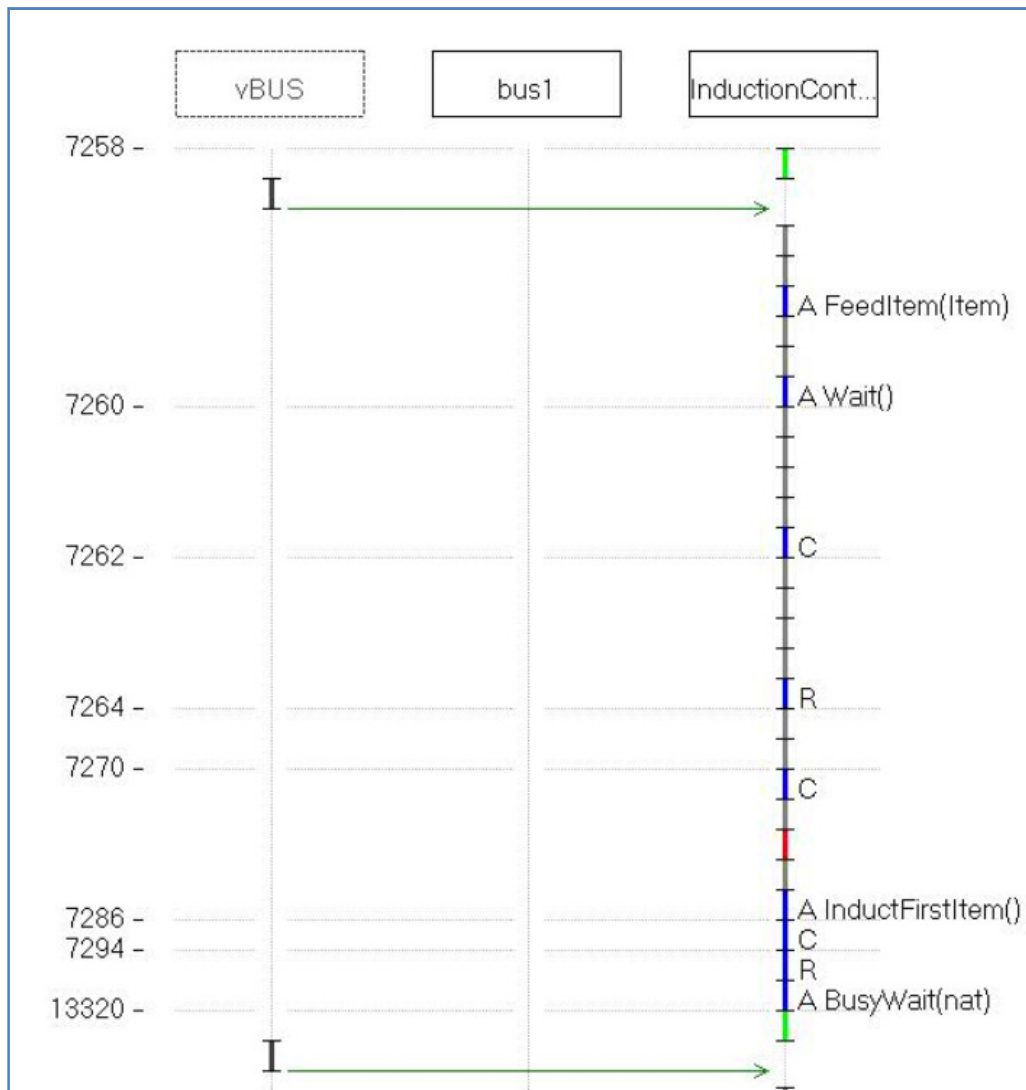


Figure 15 First slow version of the RT model with BusyWait

The problem of tray objects deployed to the vCPU was related to the creation of objects in the constructor of the TrayAllocator. In the RT model the TrayAllocator was created as static by the system class SC and all objects created by the constructor will be deployed to the vCPU since the `cpuTA4.deploy(allocator)` is called after creation of the allocator. Therefore we added the operation `CreateAllocatorObjs` that is now called by thread in the SorterEnvironment.

This modification gave us a new problem where we got a runtime error (Overture bug to be fixed) when we tried to open the log files. Therefore we will not be able to show a graphical view of log files for the final model where Trays are deployed to the cpuTA4. In the below lines that is part of the log file we can see that the new Induction class and Tray is running on cpunm: 3 which for where the TrayAllocator is deployed.

```
DeployObj -> objref: 1 clnm: "InductionController" cpunm: 1 time: 0
DeployObj -> objref: 2 clnm: "InductionController" cpunm: 1 time: 0
DeployObj -> objref: 3 clnm: "InductionController" cpunm: 2 time: 0
DeployObj -> objref: 4 clnm: "InductionController" cpunm: 2 time: 0
DeployObj -> objref: 5 clnm: "TrayAllocator" cpunm: 3 time: 0
OpRequest -> id: 16 opname: "GetId()" objref: 30 clnm: "Tray" cpunm: 3
async: false time: 1666
OpRequest -> id: 16 opname: "Induction(nat)" objref: 42 clnm:
"Induction" cpunm: 3 async: false time: 1708
```

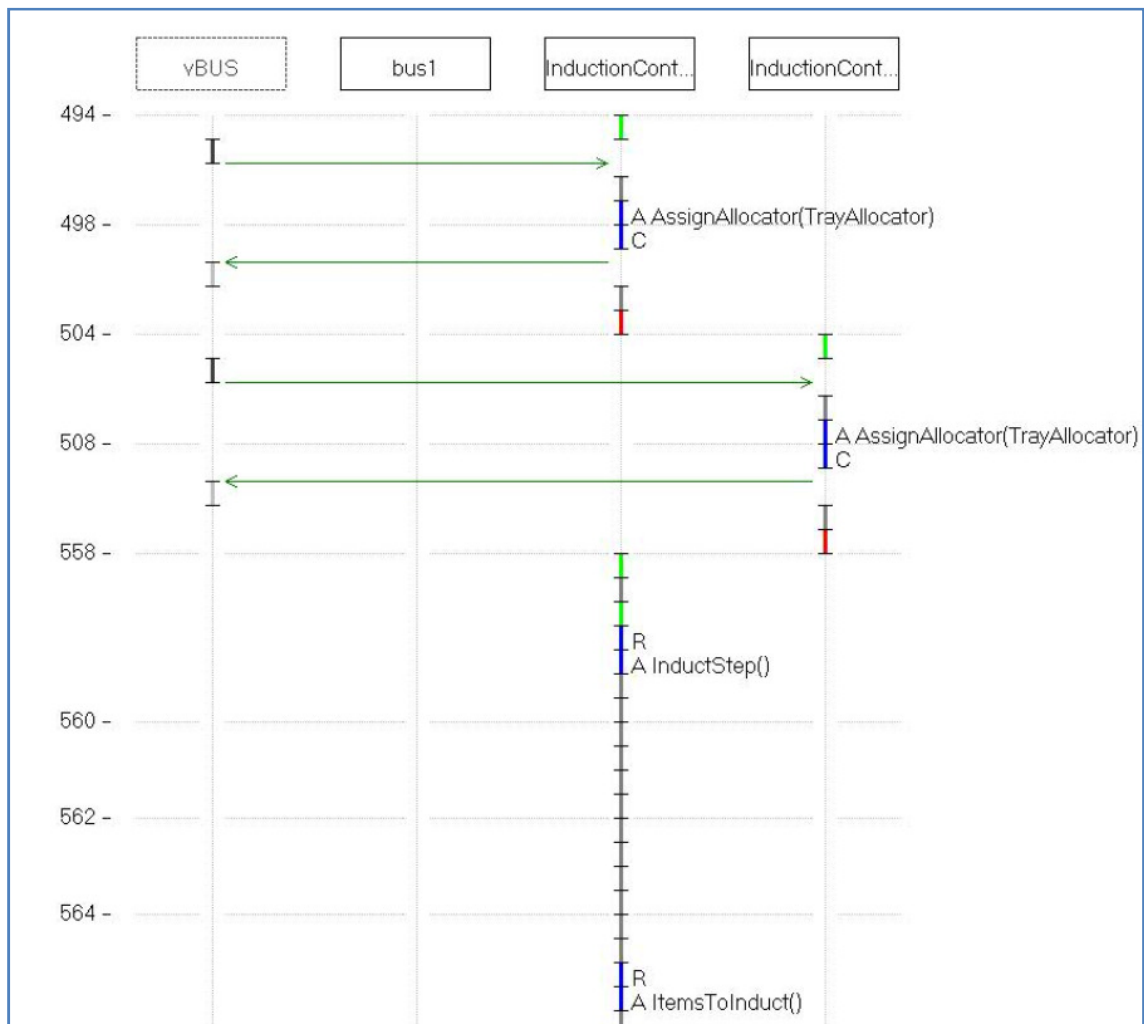


Figure 16 Final RT model deployment scenario 2

The final model still have one know issue for optimization that is related to item. In the UML diagram below shows that both the Tray and InductionController is associated to an Item. The final RT model is change so the InductionController creates new Items but only a reference is given to the TrayAllocator. That means every time the simulation is running we still have a number of unnecessary communication between cpuIC and cpuTA.

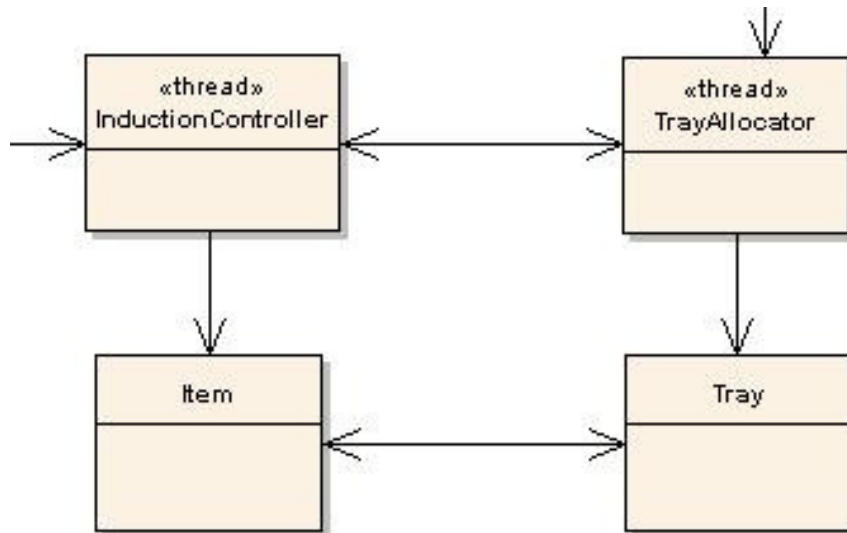


Figure 17 UML diagram for Item and Tray associations

We have tried to optimize and solve this issue by creating a copy of Items on in the TrayAllocator by change of the RequestTray operation in TrayAllocator to pass the itemid and itemsize instead of the item object see below. When we run the model we get an error on the precondition for creating the new item that the itemsize is invalid even it is not checked by the debugger.

```

public RequestTray: nat * nat * nat1 * nat ==> ()
RequestTray (tid, icid, itemid, itemsize) ==
let ic = inductionGroup(icid)
in
    let item = new Item(itemid, itemsize)
    in
        AddItem(tid, ic, item)
pre icid in set inds inductionGroup
and tid not in set dom itemsToInductMap;
  
```

Precondition failure: pre_Item in 'TrayAllocator' at line 175:20

By removing the invariant permission predicate (SizeLimits) in the Item class we did manage to simulate the model and got the same results but with a faster simulation since now we reduced the number of calls from cpuTA to cpuIC's.


```

size : nat1;
inv SizeLimits(size);

operations

public Item: nat1 * nat ==> Item
Item(s, i) ==
(
    size := s;
    sizeOfTrays := size div Tray`Size + 1;
    id := i;
)
pre SizeLimits(s);
functions

SizeLimits: nat -> bool
SizeLimits(s) ==
    s >= MinSize and s <= MaxSize;

```

The following list shows the final simulation results for the RT models. The realtime model with deployment scenario 1 (RTD1) and scenario 2 (RTD2) with a scenario file inducing items on all four inductions see appendix A for more details.

```

mk_(6900, [mk_(100,1,100),
            mk_(200,2,200),
            mk_(300,3,300),
            .....
            mk_(2400,4,300),
            mk_(2500,1,100),
            mk_(2600,2,200)])

```

We have removed all the detailed logs due to problems with overture calling static operations from threads running deployed on CPUs different from the virtual CPU. Instead we are only logging items feed by the environment with information about the time in ms, induction id, item size and simulation time tick. We see that we get the same results for both deployments and we are not able to fill the sorter. As expected there is no difference between the two deployments method, since there is no difference in the amount of communication between the different CPUs in the deployment scenario 1 and 2. All communication is between the IC and TA. If there was interaction between the ICs we would have expected a difference.

```

-----
Tray allocation RTD1 model #1 : scenario1.txt
-----

[ 100 , 1 , 100 , 6214 ]
[ 200 , 2 , 200 , 8546 ]
[ 300 , 3 , 300 , 10878 ]
.....

```

```

[ 2400 , 24 , 300 , 54632 ]
[ 2500 , 25 , 100 , 55778 ]
[ 2600 , 26 , 200 , 58172 ]

-----
Simulation completed for sorter configuration
-----

Specified throughput [items/hour]: 10000
Sorter speed          [mm/sec]: 2000
Item max size         [mm]: 1500
Item min size         [mm]: 100
Tray size             [mm]: 600
Number of trays       : 20
Number of inductions  : 4
Induction rate        : 2
Induction separation   [trays]: 2

-----
Number of trays with items      : 18
Two tray items on sorter       : 1
Number of tray steps           : 23
Number of inducted items       : 17
Calculated throughput[items/hour]: 9391

-----
      ****  Sorter is not full  ****
-----

```

Simulation result deployment on 4 CPU's (Deployment scenario 1)

```

-----
Tray allocation RTD2 model #1 : scenario1.txt
-----

[ 100 , 1 , 100 , 6214 ]
[ 200 , 2 , 200 , 8546 ]
[ 300 , 3 , 300 , 10878 ]
.....
[ 2400 , 24 , 300 , 54632 ]
[ 2500 , 25 , 100 , 55778 ]
[ 2600 , 26 , 200 , 58172 ]

```

```
-----  
Simulation completed for sorter configuration  
-----
```

```
Specified throughput [items/hour]: 10000  
Sorter speed          [mm/sec]: 2000  
Item max size         [mm]: 1500  
Item min size         [mm]: 100  
Tray size             [mm]: 600  
Number of trays       : 20  
Number of inductions  : 4  
Induction rate        : 2  
Induction separation  [trays]: 2  
-----
```

```
Number of trays with items : 18  
Two tray items on sorter   : 1  
Number of tray steps      : 23  
Number of inducted items   : 17  
Calculated throughput[items/hour]: 9391  
-----
```

```
***** Sorter is not full *****  
-----
```

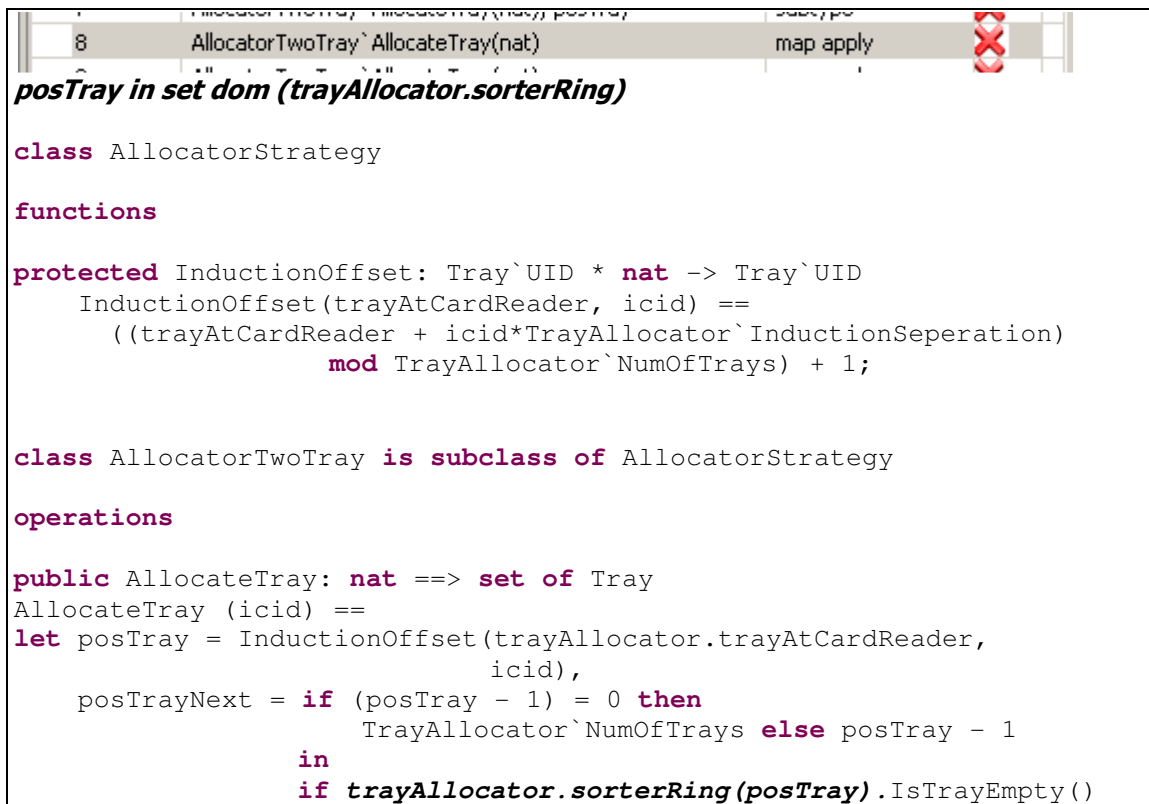
Simulation result deployment on 2 CPU's (Deployment scenario 2)

6. Proof obligation and Test coverage

6.1. Proof obligation

In ensuring the quality for the internal consistency of the model we have used proof obligation. Besides proof obligation the overture tool handles the syntax, type checking and model debugging and execution. There are approx. 80 proof obligations in the models we have made where we have investigated some of them. In the following lines we will go through some of these examples and explain where the model could be improved.

The operation AllocateTray is using a tray position modified by the InductionOffset function. It must be assured that before the AllocateTray operation is invoked, the position that is going to be used is a member of the set defined by the domain of the SorterRing map.



```

8      AllocatorTwoTray`AllocateTray(nat)      map apply
posTray in set dom (trayAllocator.sorterRing)

class AllocatorStrategy

functions

protected InductionOffset: Tray`UID * nat -> Tray`UID
  InductionOffset(trayAtCardReader, icid) ==
    ((trayAtCardReader + icid*TrayAllocator`InductionSeperation)
      mod TrayAllocator`NumOfTrays) + 1;

class AllocatorTwoTray is subclass of AllocatorStrategy


operations

public AllocateTray: nat ==> set of Tray
AllocateTray (icid) ==
let posTray = InductionOffset(trayAllocator.trayAtCardReader,
                             icid),
  posTrayNext = if (posTray - 1) = 0 then
    TrayAllocator`NumOfTrays else posTray - 1
  in
  if trayAllocator.sorterRing(posTray).IsTrayEmpty()

```

The proof obligation is indicating that the sequence of items can be an empty sequence. If this happens, a run time error will appear when the operator head is executed over items. A precondition should be added to the model stating that the length of the sequence items should be greater than zero.

```

21 InductionController`GetSizeOfWaitingItem(), item non-empty sequ... 
items <> []


inv priority > 0 => len items > 0;

public GetSizeOfWaitingItem: () ==> nat
GetSizeOfWaitingItem() ==
  if not IsItemWaiting()
  then
    return 0 -- No waiting items
  else
    let item = hd items
    in item.GetSizeOfTrays();

```

In the third case, there is a slight difference between the declared data type that is representing the induction id and the one that we are using in the InductionController constructor. In this last case this function is receiving two parameters, a reference to an allocator and a natural number (type nat). The induction id, by the other hand, is defined as a nat1. So the main problem is that a value 0 could be passed as a parameter to the InductionController constructor, violating the data definition of induction id. The solution to this problem would be to define the parameter as a nat1 as well.

```

16 InductionController`InductionController(TrayAllocator, ... subtype 
n > 0

id : nat1;

public InductionController: TrayAllocator * nat ==> InductionController
InductionController(a, n) ==
(
  allocator := a;
  id := n;
  selfIC := self;
);

```

6.2. Real Time version test coverage summary

The test coverage is offering an accurate measurement of how much the code has been exercised during the model execution. The Overture tool is running the coverage engine through all the source files and going through all the functions and operations the class presents. Finally a set of tables (one per source file) will be presented, displaying the name of the function or operation, the coverage percentage and the number of times the function or operation has been invoked. In the case that the function has never been called, it will be remarked in the test coverage output. An example of the test coverage result can be seen in the following table, which corresponds to the Tray class.

Function or operation	Coverage	Calls
GetId	100.0%	2580
GetItem	100.0%	36
GetState	100.0%	11
IsTrayEmpty	100.0%	53
IsTrayFull	100.0%	20
ItemOnTray	100.0%	18
SetState	0.0%	0
Tray	100.0%	20
Tray.vdmrt	84.0%	2738

All the operations except the SetState have been executed a number of times. The SetState operation was introduced in the model but finally it was never used, so it could be removed from the model without affecting it. In the last row of the table, an average coverage percentage is presented, with the total sum of calls.

In the next table, the information regarding the coverage percentage and the number of calls for each file in the Real-time² deployment scenario 2 is presented below:

VDM++ Class	Coverage	Calls
AllocatorStrategy.vdmrt	100.0%	36
AllocatorOneTray.vdmrt	100.0%	101
AllocatorTwoTray.vdmrt	97.0%	7
Induction.vdmrt	77.0%	1327
InductionController.vdmrt	100.0%	210
Item.vdmrt	82.0%	280
ItemLoader.vdmrt	100.0%	1283
SC.vdmrt	100.0%	1
SorterEnviroment.vdmrt	100.0%	4
Tray.vdmrt	84.0%	2738
TrayAllocator.vdmrt	100.0%	243
World.vdmrt	98.0%	2

² Coverage test files to be found on the CD for the overture RT projects named trayAllocRTOptAlt\Results\trayAllocRTOptAlt.pdf

7. Conclusions

We have realized in this project that modelling is a powerful tool in helping us to evaluate different ideas applied to the problem being studied. While analyzing the situation we came up with several strategies that have been modelled using UML and VDM++. It was possible to determine whether the solutions to the problem were a correct approach or not and the advantages and disadvantages they presented.

In this project we have moved, from a sequential model of the system, to a concurrent and further on to a real time and distributed model. This implies that we have moved from an approach in which we were focusing on logic to a perspective closer to reality representing time, concurrent threads, CPU's and buses. Since several entities that simultaneously exist and interact with each other are present in reality, a concurrent model was a must-have view. In this second approach we focused on which entity classes should be running in parallel and how they were going to be synchronized.

Even though the concurrent model is a better model of the system, it is still important to consider issues like, how is it going to behave in real time or how is it going to be deployed in the real scenario of deployment. In order to cope with these two last issues, the real-time and distributed model introduced mechanism to deal with scheduling and facilities to simulate different deployments.

As it can be seen in this modelling project has experienced a progression, going from a purely logical model to a model that could be used to extract information and conclusions about how a real deployment should be performed. For example, the required busses bandwidth or the throughput the different used CPU's could be. Both of them are parameters that can be analyzed for a given topology. The other way around is possible as well, and that is, which topology suits the best this scenario? By considering this, it is possible to use our VDM model to experiment with different deployments and determine which structure is the optimal one.

By analysing the performance of the system after running several tests, some feedback can be obtained. This feedback can be taken as an input in the development process and used to improve the model of the system, implying that the model quality will be improved and the system as well.

The progression the project has experienced has had a positive impact in the model quality. At each stage, each model has been given us information about different topics. For example, the sequential model allowed us to find the best logical approach for the problem. The concurrent model allowed us to find the best way of modelling the concurrent nature of the system. Finally the real time distributed model gave us insight in how we could split the design and deploy the

functionality of the model to reduce the interaction between the different CPU's in the system. Even we did have problems with some of the errors in the overture VDM++ tool it has been very helpful in archiving our goal. It is important to remark that this three staged process is quite logical, since it is almost impossible to go straight to a real time model. We have found it helpful to use this stepwise approach starting with a sequential logical model and secondly continue with a concurrent or perhaps continue directly with the real time distributed model depending on the goal of the modelling project.

For the distributed model since different functionalities are mapped to different processors in the considered deployment scenarios, we are able to experiment on how the components interact with each other and how the information is flowing between different objects. This can have a direct impact on the system performance, and as it has been explained in the previous chapters, minor changes can affect the whole system. We would recommend extending the overture VDM++ tool to be much better at this point in providing functionality like graphical zooming, searching and filtering the log files especially for analysing very complex RT models.

As a general conclusion, it should be said that debugging, analyzing and improving distributed systems is a tough labour, and tools like abstraction, graphical representations, UML diagrams and formal methods are the only way to cope with the inherent complexity of this field.

8. References

- [1] Bjerger, Kim. Esparza Isasa, José Antonio. "Tray allocation for a sortation system" Model driven development using VDM++ and UML. Spring 2010.
- [2] Larsen, Peter Gorm. Lausdahl, Kenneth et al. "VDM-10 Language Manual" April 2010.
- [3] Fitzgerald, Larsen et al. "Validated Designs for Object-Oriented Systems" Springer 2005.
- [4] Larsen, Peter Gorm. Slides and materials from the course "Model driven development using VDM++ and UML 1" Aarhus Engineering College. Spring 2010.
- [5] Larsen, Peter Gorm. Slides and materials from the course "Model driven development using VDM++ and UML 2" Aarhus Engineering College. Spring 2010.
- [6] Coloured Petri Nets: brief introduction from the Computer Science Department from Aarhus University. <http://www.daimi.au.dk/CPnets/intro/> [Link valid April 2010]
- [7] "Overture – Open-source Tools for Formal Modeling".
<http://www.overturetool.org/twiki/bin/view> [Link valid April 2010]
- [8] Viena Development Method: Wikipedia article on VDM. General information, code examples and references. http://en.wikipedia.org/wiki/Vienna_Development_Method [Link valid April 2010]
- [9] Larsen, Peter Gorm et al. "Development of Distributed Embedded Systems using VDM" April 2010.