

Tray allocation for a sortation system

Model driven development using VDM++ and UML

Spring 2010



Kim Bjerger kbe@iha.dk 20097553

José Antonio Esparza jaesparza@gmail.com 20097706

Table of contents

1.	Introduction	3
1.1.	Report structure	3
2.	Problem Description	4
2.1.	System description.....	4
2.2.	System requirements	6
2.3.	Purpose of the model.....	7
2.4.	Terminology	8
2.5.	The system from a discrete perspective	9
2.5.1.	Preliminary considerations.....	9
2.5.2.	Working example.....	9
3.	Possible allocation algorithms	12
3.1.	The pattern matching allocation algorithm	12
3.2.	The priority scheme algorithm	12
4.	System architecture	13
4.1.	The distributed architecture.....	13
4.2.	The centralized architecture	14
5.	Model of the system	16
5.1.	Overview of the distributed and centralized model in UML.....	16
5.1.1.	The distributed model	16
5.1.2.	The centralized model	17
5.1.3.	The chosen architecture	18
5.2.	The centralized model in UML and VDM	18
5.2.1.	The Tray class	21
5.2.2.	The Induction Controller class	23
5.2.3.	The Tray Allocator class.....	25
5.2.4.	The strategy pattern applied to the tray allocator	31
6.	Model system testing	35
7.	Conclusion	43
8.	References.....	45

Appendix A: The pattern matching algorithm

Appendix B: Transcript of the VDM++ model

Appendix C: Test results

1. Introduction

The main purpose of computer based systems is to help stakeholders dealing with repetitive or complex problems that may occur daily in their surroundings. An important matter from a computer perspective is dealing with this complexity in order to offer a reliable and appropriate solution to the stakeholders problem, something that can be complicated in most situations.

In order to cope with this issue some methods and tools can be used like, for instance, modelling. By modelling a certain problem or situation, some conclusions and further information can be extracted, helping designers and engineers to create new solutions that can be validated from the very beginning of the development process.

Applying this idea to the software construction field, it can be conclude that combining modelling with formal methods and languages under the object oriented paradigm could be an extremely helpful way to construct reliable software that suits the real problem. In this way VDM++ is a perfect tool that offers all the above exposed features to help on the system analysis and development tasks.

A current problem that presents both repetitive and complex features is the tray allocation for a sortation system. In this kind of systems several strategies can be applied in order to optimize performance and obtain a given system throughput. In order to reach the most suitable strategy preliminary studies and models should be created. By using VDM++ in this context, we will be able to test a certain strategy over a set of scenarios, before even starting to implement a single line of code or requiring any kind of specific hardware or software.

1.1. Report structure

This document is composed by six main parts, in which the overall system and designed model will be introduced. The document starts with an introduction, explaining what the project is about and a brief approach to the purpose of applying VDM++ and UML modelling. The problem description will go in further details introducing a more extensive description, presenting the system requirements and the used terminology that has been kept along this project. After that, possible allocation algorithms that can be applied will be presented. Several architectural solutions will be introduced at the system architecture chapter, being compared and discussing their advantages and disadvantages. The model of the system section will present the system from both UML and VDM++ perspective making strong emphasis in the logical composition of the model and how it evolves with time. In the section model system testing some analysis of how the model works and which kind of information can we obtain from its execution will be found. To end up some conclusions and reflections will be commented in the conclusion section.

Additional information about this work will be attached as appendixes to this report, mainly containing a transcript of the full VDM++ model, test scenarios and the Description of a possible allocation strategy that have not been used in our model.

2. Problem Description

2.1. System description

Sortation systems are used for post and parcel sortation in post centres or at airports for baggage sortation. The system consist of a number of conveyors that transports items (post packages and letters) to the central part of the sortation system that consist of a number of trays that sorts items to different discharges from where they are transported to a final destination. The system overview is illustrated in the figure below.

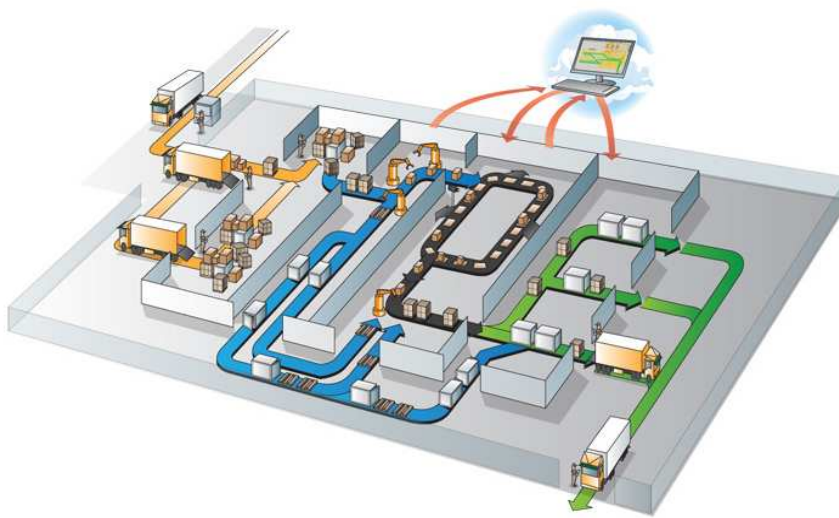


Figure 1: Sortation system at post centre [10]

The considered problem is about this central sorter part of the system that is able to transport several items placed on a tray in the sorter ring. The items are fed to the sorter ring by several inductions located around and grouped in induction groups. In the figure below it is possible to see four inductions feeding items to the sorter ring.



Figure 2: Several inductions (left) and sorter ring [10]

Since feeding the items to the sorter should be done automatically the system should be able to avoid certain situations that can lead to poor performance or even system malfunctioning. Here some basic ones are introduced:

- **Two items placed on the same tray:** it is physically possible, but this situation could end delivering one of the items to an erroneous discharger.
- **A poor sorter usage:** in the case the inductions are not inducing the items properly for instance delaying too much the induction because of an improper allocation policy. This can imply that a considerable percentage of the sorter trays are empty without transporting any items.
- **Induction starvation:** since the sorter ring should transport items with a variable size the allocation strategy should be aware of reserving some empty trays for bigger items, but this does not mean that the same induction always has to wait in order to leave an empty tray.

So it can be concluded that finding a proper allocation strategy is the key point in order to get an overall good system performance. It should be considered as well that there are numerous variables that can influence this result. Some of them could be the speed of the sorter, the target throughput or the different parcel sizes that the system should handle.

To study the allocation problem we have focused on a system in which three inductions are trying to induct items to a sorter ring. These three inductions belong to the same induction group. Before the induction group a card reader is located being able to determine the identification of the tray and if it is empty. This is the key position to determine the trays passing in front of the inductions, based on the offset between the inductions the card reader.

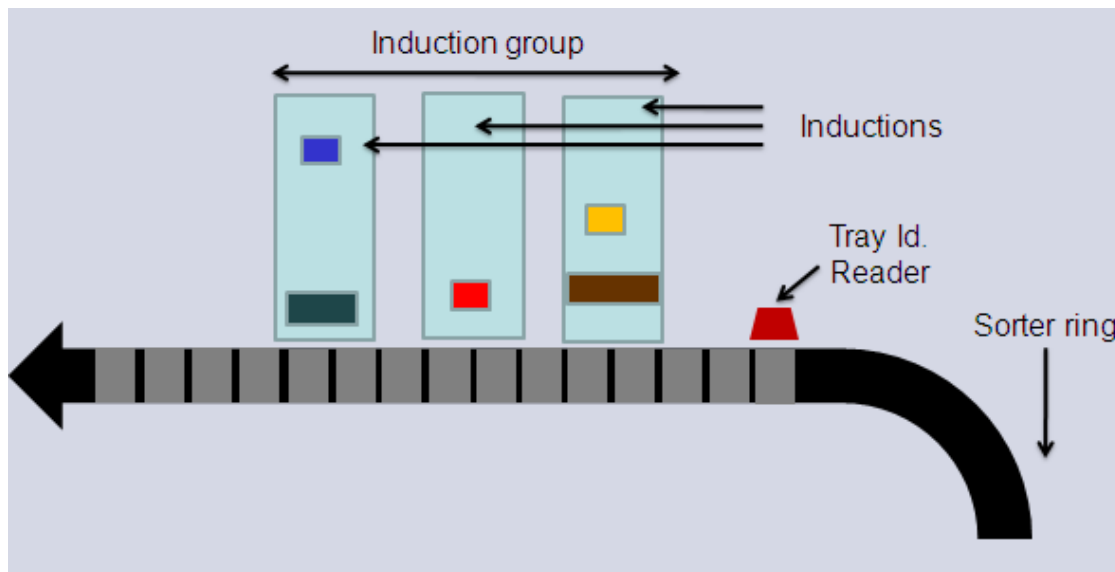


Figure 3: A simplified sorter model

Once the simplified model was outlined, the next important step is to obtain a set of system requirements that the system should meet.

2.2. System requirements

After analysing the system through the first two iterations of the project, the following requirements were found:

1. The sorter-ring moves at a constant speed.
2. The sorter-ring must be composed by a number of trays with a certain size.
3. The trays must be consecutively located at the sorter-ring.
4. The system should assign a unique identifier per tray (trayUID).
5. Each tray must be occupied by one item at most.
6. The allocation algorithm must handle the situation in which a tray is already occupied.
7. The system should handle items with different sizes, from a minimum to a maximum.
8. The system must be able to work at a maximum throughput, enabling the inductions to feed items at a maximum speed.
9. The system shall handle inconstant items arrival rate at the inductions.
10. The maximum item size should fit on two trays.
11. The system should be capable of working with several inductions per induction group.
12. Each induction group should have a separate card reader.
13. The system should be able to handle sorter capacity based on throughput and item size, in order to compute the tray allocation algorithm.
14. The tray allocation algorithm shall not cause starvation of the inductions.
15. The allocation algorithm must allocate empty trays for the inductions.
16. The dischargers should be able to empty the sorter ring at any given configuration.

The above exposed requirements can be classified in different groups, **physical requirements** (1,5,10), **construction requirements** (2,3,11,12), **behavioural requirements** (6,7,8,9,16) and **logical requirements** (4,13,14,15). The physical requirements are inherent to the physical nature of the system (for example, it is impossible to occupy an already occupied tray) and influences the construction requirements. All kinds of requirements should be considered in order to analyze their impact on the behaviour and the logic of the system.

Some of the requirements introduce values and variables, like the parcel size, that will belong to a range between a maximum and a minimum value. Another value and variable is the required and calculated throughput that determines the number of items per hour that can be sorted. Both parameters should be customizable by the customer, so our model can suit several situations.

2.3. Purpose of the model

Creating a model of the system will be beneficial for the development process due to the following reasons:

- It gives a flexible framework to simulate different operative conditions. Any parameter that has been configured at the model can be altered and modified and its consequences can be evaluated. For example the model can be easily reconfigured to work with more inductions or with a certain arrival rate.
- It acts as a mechanism to evaluate the performance of different allocation strategies. In the case that different allocation strategies are being considered they can be modelled in order to test them.
- Allows the designers to have a tool to determine the validity of a given hypothesis concerning the system. Any consideration regarding system configuration can be modelled and discussed.
- Allows automatic code generation. This can speed up the development process in the case a real system should be delivered spending less time in the implementation and testing phase.

2.4. Terminology

Since the first iteration of the project, it was decided that it was highly important to have a common and constant naming convention and to maintain it along the whole project. In the following lines the used terminology is listed.

- **Card reader** Device that reads the unique identifier of the trays located at the induction group.
- **Discharger** Mechanical device responsible for taking out items from the sorter ring.
- **Induction** Mechanical device inducing items on the sorter-ring.
- **Induction controller** Mechanism responsible for controlling the basic operating signals that handles an induction. Communicates with a central controller in the case it is being used.
- **Induction group** A group of induction capable to fill the sorter-ring with items.
- **Induction separation** Number of trays between each induction. At minimum the tray separation is one otherwise it will not be feasible to operate the sorter.
- **Item** Item (post letters, packages, books or baggage ect.) that is fed by the induction to the sorter and transported until it is discharged.
- **Priority** Numerical value that determines the weight of the induction request in order to be considered by the controller. The priority influences if should be inducted or not.
- **Sorter** The complete sorter system that consist of trays, inductions and discharges (Machine and control)
- **Sorter controller** (SC) Mechanism responsible for managing several induction controllers. In the case the SC is responsible as well for allocating the items on trays.
- **Sorter ring** Ring of trays that transports parcels from the induction group to the discharges.
- **Tray** Element that transports and sorts items.
- **Throughput** Items per hour that can be handled by the sorter.

2.5. The system from a discrete perspective

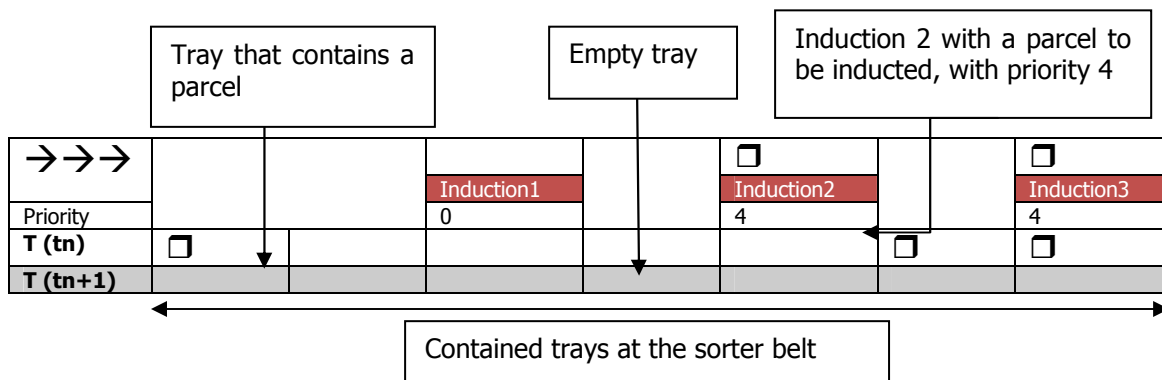
Considering a system like the one presented in figure 3 a basic 9 step operation will be explained in the following lines. This description is based on the *priority scheme algorithm* described in detail in the next chapter.

2.5.1. Preliminary considerations

In order to describe the algorithm from a simple and meaningful point of view some assumptions have been made:

- The Induction Separation has been set to 1.
- The Priority has been set to 0 at the initial state. That means that each induction is considered at the same level in order to obtain an empty tray.

From now on, the part of the system that is being studied will be represented as follows:



2.5.2. Working example

Initial state

After system initialization the trays located at the sorter ring are empty and the inductions that compose each induction group have no parcels to induct to the sorter.

This is state will be represented as follows:

→→→						
		Induction1		Induction2		Induction3
Priority		0		0		0
T (tn)						
T (tn+1)						

State 1: time tn

In this example 3 items arrive to the induction group, one per induction. When the items have been assigned to an empty tray they are inducted to the sorter.

At the beginning of the state, the system will be:

→→→		□		□		□
		Induction1		Induction2		Induction3
Priority		0		0		0
T (tn)						
T (tn+1)		□		□		□

State 2: time t_n+1

At this state, the items inducted at State 1 have advanced 1 position due to the tray step. Three new items arrive to the inductions, and since there are no items on the sorter ring blocking them, they can be inducted to the sorter.

At the beginning of the state:

→→→			<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>
			Induction1		Induction2		Induction3
Priority			0		0		0
T (t_n+1)				<input type="checkbox"/>		<input type="checkbox"/>	
T (t_n+2)			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

State 3: time t_n+2

At the beginning of the state, the sorter has advanced one tray step and three new items have arrived to the inductions. It is only possible to induct the item located at the induction 1, so items contained at induction 2 and induction 3 will have to wait and the priority is incremented.

→→→			<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>
			Induction1		Induction2		Induction3
Priority			0		1		1
T (t_n+2)				<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
T (t_n+3)			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Considering no more arrivals to the inductions the system will evolve as follows:

State 4: time t_n+3

→→→			<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>
			Induction1		Induction2		Induction3
Priority			0		2		2
T (t_n+3)				<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
T (t_n+4)				<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

State 5: time t_n+4

→→→			<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>
			Induction1		Induction2		Induction3
Priority			0		3		3
T (t_n+4)					<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
T (t_n+5)					<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

State 6: time t_n+5

→→→			<input type="checkbox"/>		<input type="checkbox"/>		<input type="checkbox"/>
			Induction1		Induction2		Induction3
Priority			0		4		4
T (t_n+5)						<input type="checkbox"/>	<input type="checkbox"/>
T (t_n+6)					<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

State 7: time t_n+6

→→→							<input type="checkbox"/>
			Induction1		Induction2		Induction3
Priority			0		0		5
T (t_n+6)						<input type="checkbox"/>	<input type="checkbox"/>
T (t_n+7)							<input type="checkbox"/>

Step 8: time t_n+7

→→→							<input type="checkbox"/>
			Induction1		Induction2		Induction3
Priority			0		0		6
T (t_n+7)							
T (t_n+8)							<input type="checkbox"/>

Step 9: time t_n+8

→→→							
			Induction1		Induction2		Induction3
Priority			0		0		0
T (t_n+8)							
T (t_n+9)							

3. Possible allocation algorithms

3.1. The pattern matching allocation algorithm

The lookahead mechanism could be defined as a tool for looking ahead a few more input items before making a cost effective decision at one stage of the algorithm. In this particular case the input items will be the trays contained at the sorter and the cost effective decision will be to feed an item into the sorter.

A lookahead buffer can be configured in order to start considering the sorter positions earlier, so the system will have more information in order to offer a better item distribution along the sorter. The algorithm will try to match an encoded induction state with the encoded sorter state. In the case that the induction pattern does not match with the positions located in front of the inductions, the algorithm can shift it to the left in order to consider the rest of the encoded sorter, taking advantage of the lookahead buffer.

Considering a bigger lookahead buffer size also implies performing more calculations in order to match the item distribution at a given time step. This algorithm is described in details in appendix A and it will not be used in the final model.

3.2. The priority scheme algorithm

The *priority scheme algorithm* is simpler than the *pattern matching algorithm* and requires less computational effort.

This algorithm states that each induction is allowed to induct an item to the sorter unless there is other induction with a higher priority requesting a tray.

A priority rule is introduced as follows: A value called priority is incremented by value n every time an induction controller has waited one tray step to induct an item. Each induction controller will have a priority value.

Combining the main algorithm procedure with the above defined priority rule, induction groups can manage themselves locally, faster and requiring less expensive hardware infrastructure.

By using this algorithm an optimal allocation will never be obtained, on the contrary a higher throughput will be reached. This is something that was specifically expressed at the problem requirements definition stage.

The simplicity of this allocation scheme has a strong impact on the system architecture, introducing remarkable advantages that will be discussed in further points.

4. System architecture

Two main system architectures have been considered as solutions to our system. In the following lines the key difference between them and how the selection influences the system from both logical and physical perspective will be discussed.

4.1. The distributed architecture

In this architecture the induction controllers communicate between them and not with a central sorter controller, a component that it is not even preset at the system. This is the main difference if it is compared with the centralized architecture (presented at the following section).

From a logical perspective, the induction controllers should negotiate between each other in order to obtain an empty tray considering the priority present at each induction. This is a direct application of the *priority scheme algorithm* that can be perfectly supported by the distributed architecture introduced at this point.

The architecture could be represented using a deployment diagram as follows:

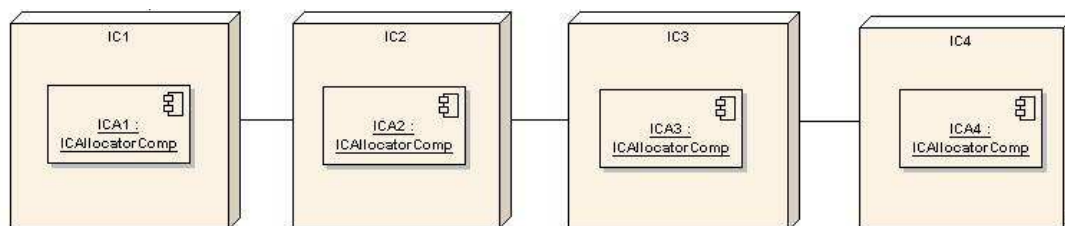


Figure 4: Deployment diagram for the distributed architecture.

Advantages:

- ✓ Reliability
- ✓ Emergent behaviour
- ✓ Easily induction disposal reconfiguration
- ✓ Components can be easily substituted
- ✓ Simpler communication layer
- ✓ Several induction groups can be easily added to the sorter without even knowing about the others

Disadvantages:

- ✗ Higher cost per induction controller
- ✗ It is complicated to get an overall optimized distribution

- ✖ It is harder to modify the used allocation strategy since it is implemented and running in several nodes. In the case a change is required; the binaries should be deployed at every machine present at the induction controllers
- ✖ The pattern matching algorithm is poorly supported

4.2. The centralized architecture

The centralized architecture is characterized due of the presence of a central sorter controller that communicates with each induction controller trough a point-to-point connection. The sorter controller will be the responsible for computing the tray allocation distribution. That means it should maintain a record of the overall sorter occupation of trays and constantly modify it in order to reflect the current sorter state.

One of the most important features of this architecture is that since the sorter controller has the big picture of the whole sorter, as well as the situation of all the inductions and induction groups, it should be able to compute all the relevant arrangements in order to obtain an optimal distribution. The centralize architecture is able to handle both *priority scheme* and *pattern matching allocation algorithm*. Nevertheless, depending on the sorter size and the number of inductions this can become a complex problem, requiring a powerful network and central computer to be solved. These issues should be considered in order to determine the feasibility of the solution at every particular case.

The architecture could be represented using a deployment diagram as follows:

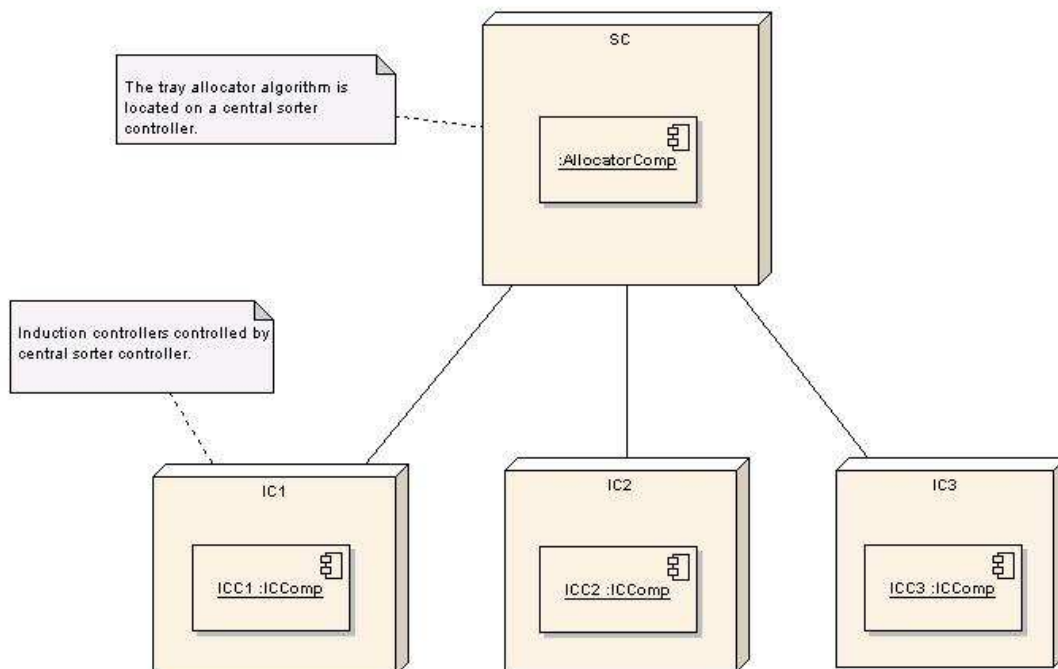


Figure 5: Deployment diagram for the centralized architecture

Advantages:

- ✓ Dummy induction controllers, simple and inexpensive hardware
- ✓ The optimal tray allocation will be obtained using this architecture, since it is possible to keep track on all the trays contained at the sorter.
- ✓ It is easier to change the allocation algorithm and allocation policies because they are defined at a single place.
- ✓ Supports both *pattern matching* and *priority scheme* allocation algorithms.

Disadvantages:

- ✗ Expensive Sorter Controller Hardware
- ✗ Bigger computing requirements
- ✗ High dependency on the Sorter Controller (Single point of failure)
- ✗ High performance communication layer required

5. Model of the system

5.1. Overview of the distributed and centralized model in UML

After considering both architectures and the different algorithms that can be applied we decided to model both systems from a UML perspective. In order to do this we used class diagrams which allow us to think about the objects, classes and relationships that compose the system including methods and attributes. The result of this preliminary design stage is reflected in the following chapter.

5.1.1. The distributed model

The distributed architecture is mainly characterized by the ability of each induction to communicate with the surrounding inductions in the group. Each induction should be able to communicate with the previous and the next induction related to it. This characteristic is present in the model in the self-relation existing in the InductionController class. In order to reflect that an induction can be located just at the beginning or at the end of the induction group (meaning that the prevIC and nextIC could be nil) both multiplicities in the relations previous and next have been assigned the range 0..1.

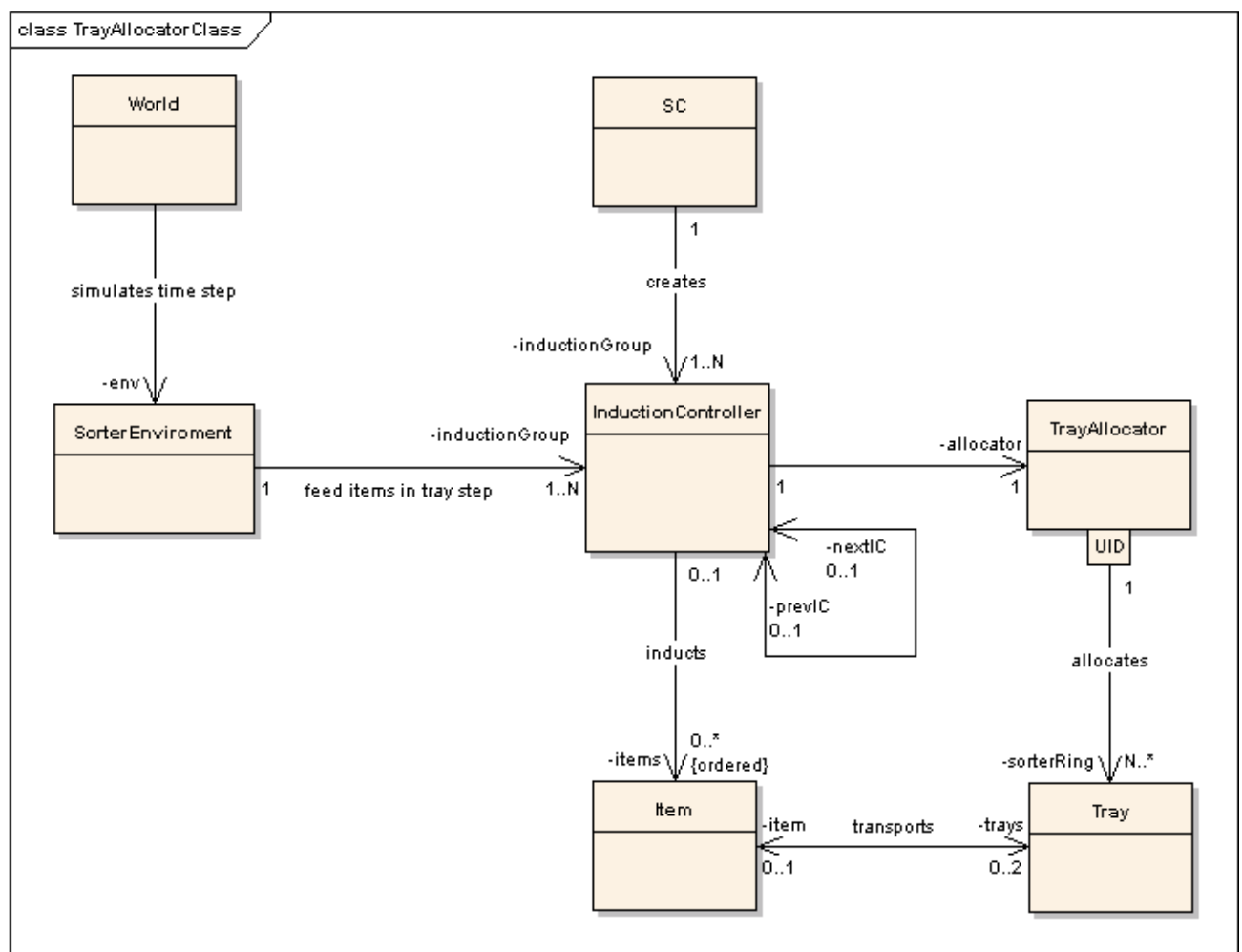


Figure 6: Class diagram for the distributed architecture system.

The Sorter Controller class (SC) is present in the system, being responsible to create and initialize the instances of the class Induction Controller with no further responsibilities. It is important to remark that the existing relation between the TrayAllocator and the InductionController class is 1 to 1. This relation is a direct consequence from the kind of algorithm that should be implemented in this architecture that is mainly focused on controlling the induction decision (Induct vs. Wait) locally to each induction, and base upon the communication maintained with the surrounding inductions.

5.1.2. The centralized model

The conceptually opposite system to the previous one, the centralized model, is represented in the below class diagram. In this case the InductionController is not required to have any notion about the possible existence of other inductions, acting as an individual element just communicated with the TrayAllocator class. In this case, the sorter controller will be responsible for notifying the tray steps to the tray allocator that is aware of the state (empty or occupied) of several trays. Another important difference regarding the TrayAllocator class is that it is communicating with several induction controllers. This is quite relevant because the TrayAllocator will act as a central point in which the association between an empty tray and an item will be created.

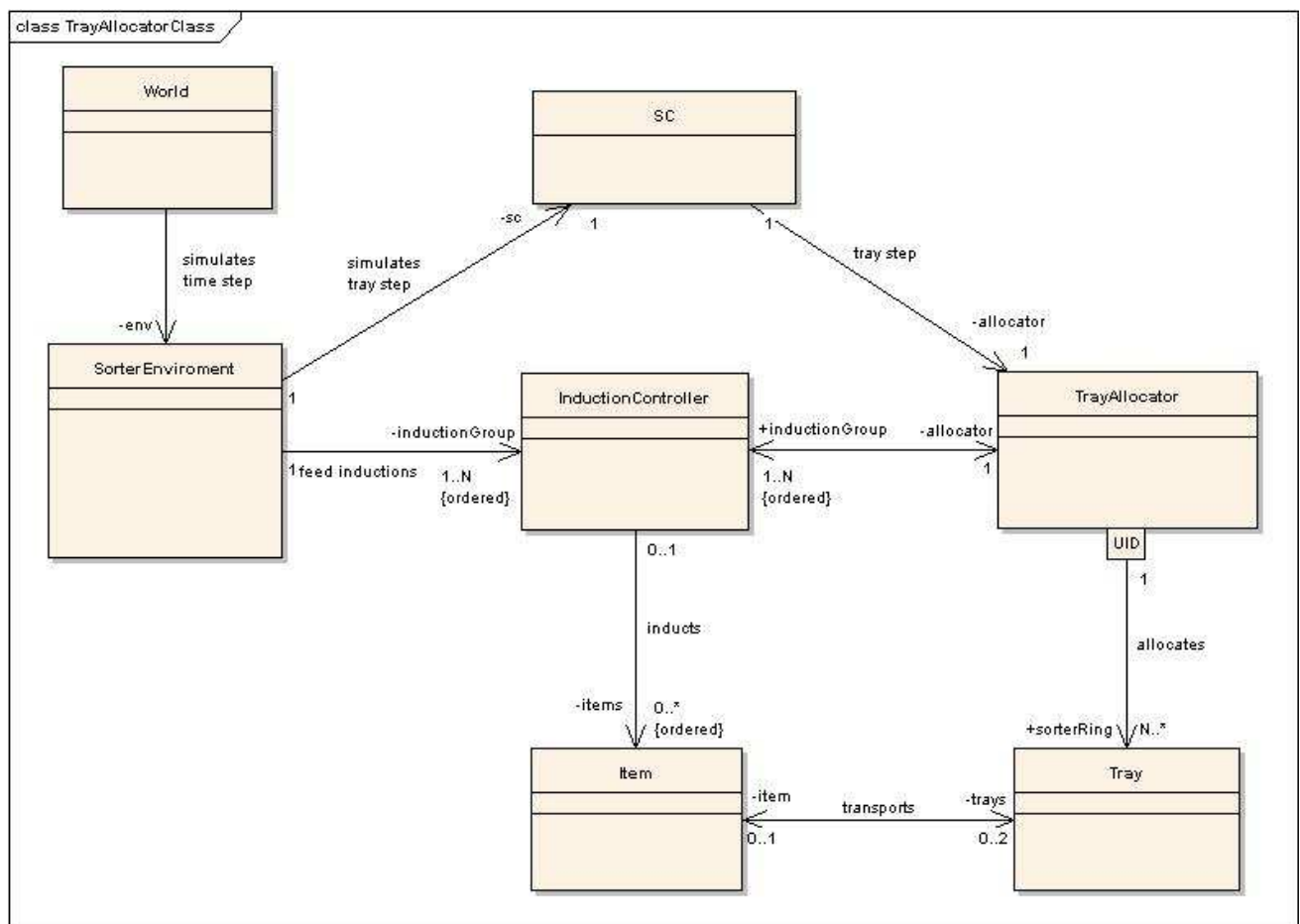


Figure 7: Class diagram for the centralized architecture system.

5.1.3. The chosen architecture

After analyzing both architectures the centralized version was finally adopted. This version allows testing a wider variety of allocation strategies, including even the ones in which a distributed architecture suits best offering in this manner a more flexible model. It is worth to mention that this is the currently used control architecture for this kind of systems, so trying to model the system under these conditions will position us closer to the real logical problem that is present in this kind of sortation systems.

The present project will be focused on modelling the sortation system from a centralized architecture applying the already exposed 3.2 The *priority scheme algorithm*.

5.2. The centralized model in UML and VDM

For better system understanding from a dynamic perspective it was considered that a sequence diagram showing how an item is inducted to the sorter should be introduced.

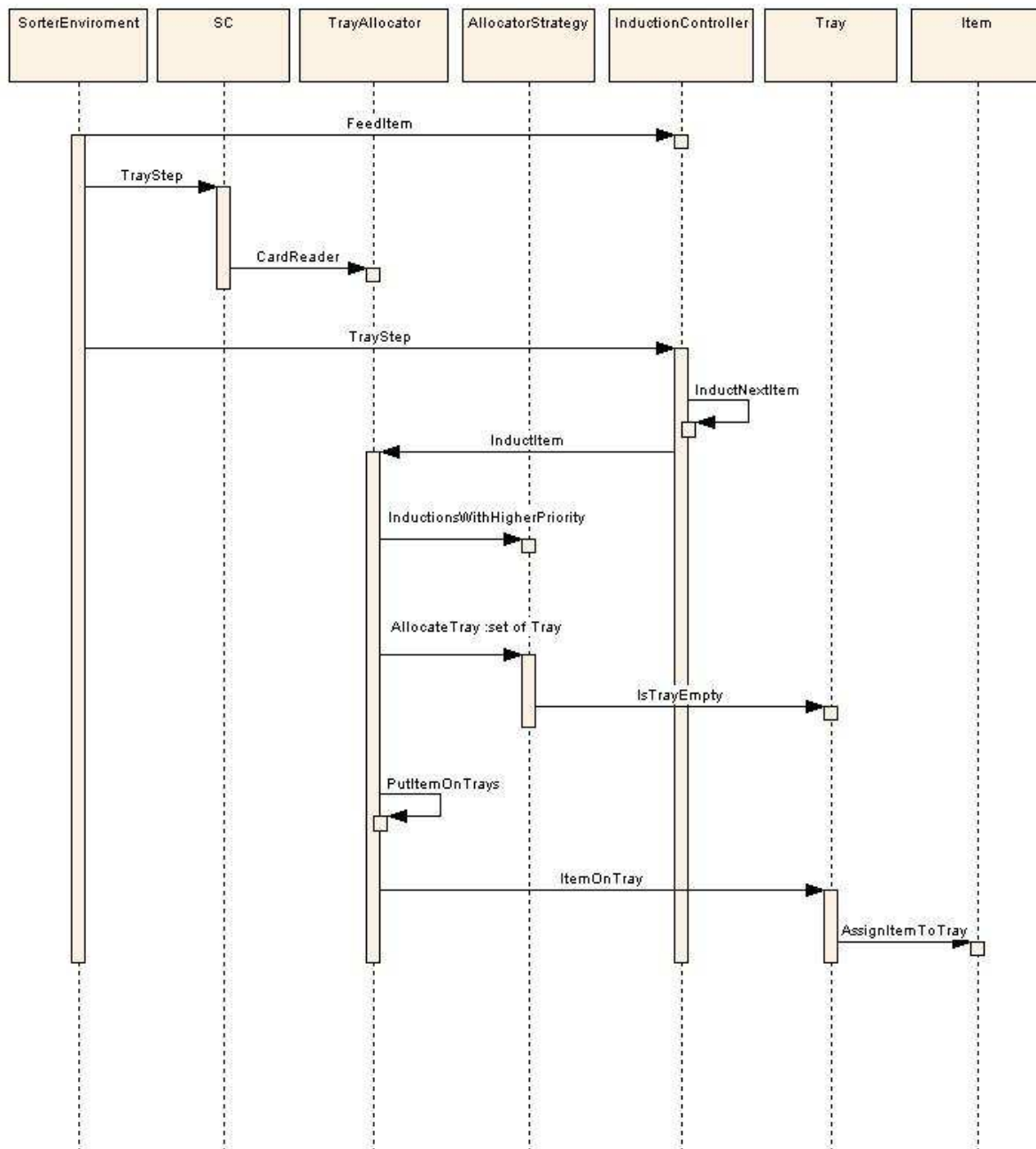


Figure 8: Sequence diagram for tray assignment and item induction

In the sequence diagram it is illustrated how an item arrives to the system and how it is inducted on to the sorter. In this particular case the item is not waiting because there is an empty tray and no prioritised inductions. Once a time step is signalled several actions in our system are triggered. The first action is that an item is fed into the induction and it is followed by a tray step in the sorter controller (SC), which means that the next tray passing in front of the card reader will be logged by the tray allocator.

The tray step will trigger a request from the induction controller to induce the next item waiting at the induction. Allocation of an empty tray will be managed by the allocator strategy. Once this step is completed the item is placed on the tray and the association between the tray and the item is created.

After some detailed analysis on the functionality, behaviour, requirement constraints and already identified classes a more detailed class diagram with concrete operations and attributes is obtained. This resulting diagram is introduced in the figure below.

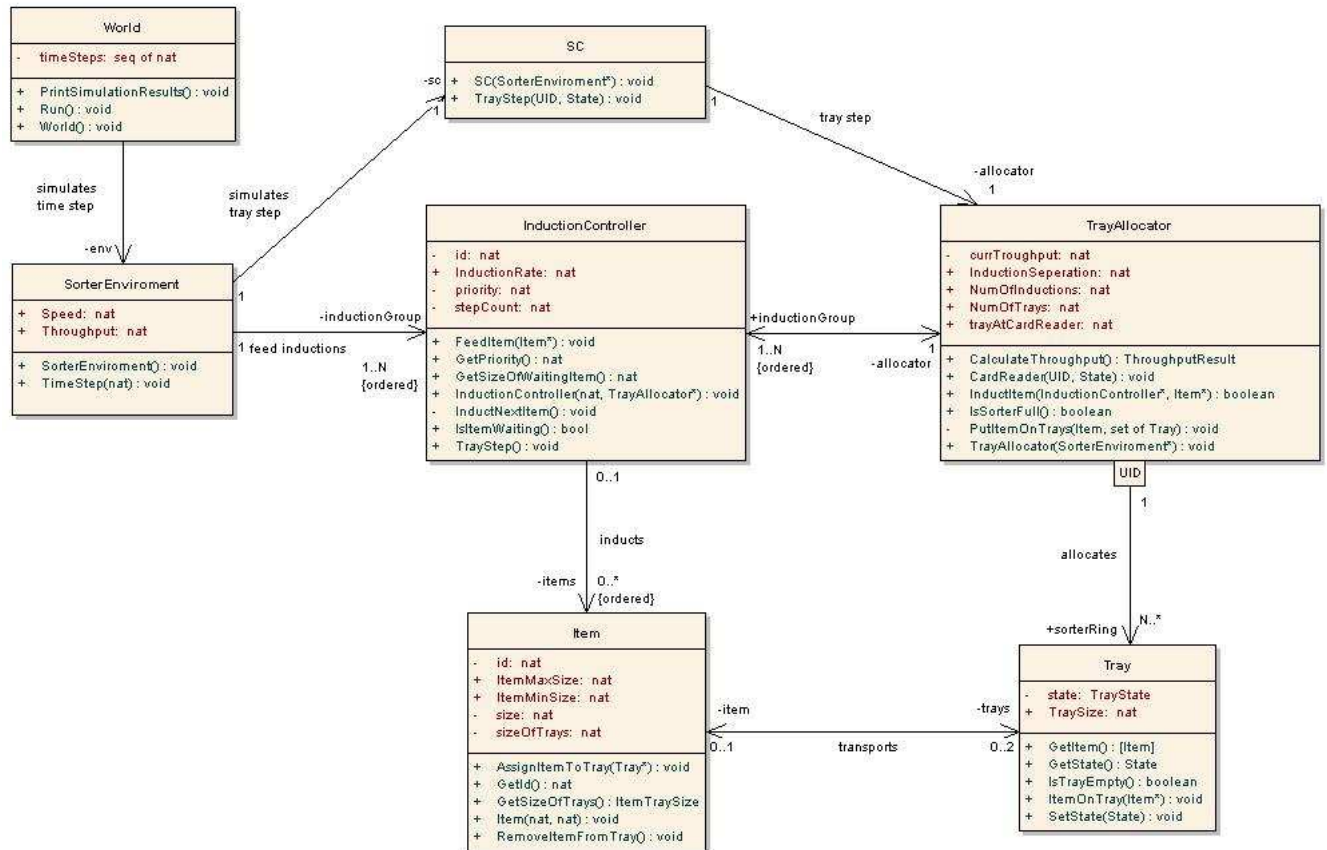


Figure 9: Class diagram with the most relevant methods and attributes

The studied system evolves with time where some reliable and constant time measurement should be introduced. This is done at the world class with the time steps. The world class will execute the system (Run) and supervise its performance (PrintSimulationResults). This is a good example of how physical movement have been abstracted. A time step is the time the sorter ring takes to move the distance equal to the length of one tray.

The World class acts as a basic object for the SorterEnvironment. The sorter environment will generate the required inputs for the model in order to operate, like for instance the tray step (moves the sorter ring to the next tray detected by the card reader) or feeding inductions with items.

In the following sections the most relevant parts of the system will be covered in detail comparing the UML class with its associated VDM++ model emphasizing the used VDM modelling that have been covered in the course.

5.2.1. The Tray class

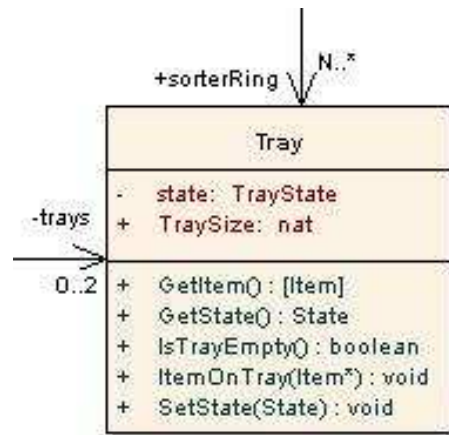


Figure 10: Detailed view on the Tray class

The Tray class represents a tray contained at the sorter. The state of the tray can be either Empty or Full which is represented in the VDM class with an enumerated type composed by two distinct enumerated values. These values are represented as an union of two quoted values: <Empty> and <Full>.

In order to associate the tray to the TrayAllocator a natural number called UID is declared as a type. The selected criteria for numbering trays do not allow assigning an UID (Unique Identifier) higher than the number of trays. This constrain is specified in the associated invariant.

```

class Tray
  types
    public State = <Empty> | <Full>;
    public UID = nat
    inv u == u <= TrayAllocator`NumOfTrays;

  values
    public TraySize : nat1 = 600
  
```

Instance variables have been declared in the Tray class for State and Item. A possible situation is that the card reader detects an unknown item this will lead to have state equal to <Full> with no item assigned. That is the reason behind expressing the item variable as optional by using []. An invariant concerning item is used to express if an item is associated with a tray the state must be full (Implication is used).

```

instance variables

    state : State := <Empty>;
    item : [Item] := nil;
    inv item <> nil => state = <Full>;

.....
operations

```

In order to assign an item to the current tray the ItemOnTray operation is specified. The enumerated variable state is assigned the value <Full>. It is necessary to use a self reference of the item to the operation AssignItemToTray. The self parameter is used to create a bidirectional association between the item and tray classes.

```

public ItemOnTray: Item ==> ()
ItemOnTray (i) ==
(
    state := <Full>;
    item := i;
    item.AssignItemToTray(self);
)
pre state = <Empty> and item = nil;

```

Below we have added an example for an implicit operation from the Item class. It specifies that it will remove the item from the current associated tray by the specified post condition. An implicit operation or function specifies what should be done and not how to do it. In our model we do not need to remove items from the sorter ring as long we only are focus on filling the sorter. That means we limit our simulation duration to the time it takes the sorter ring to move one round (Simulation time steps ~ number of trays on sorter ring).

```

public RemoveItemFromTray ()
ext wr trays : set of Tray
post trays = {};

```

5.2.2. The Induction Controller class

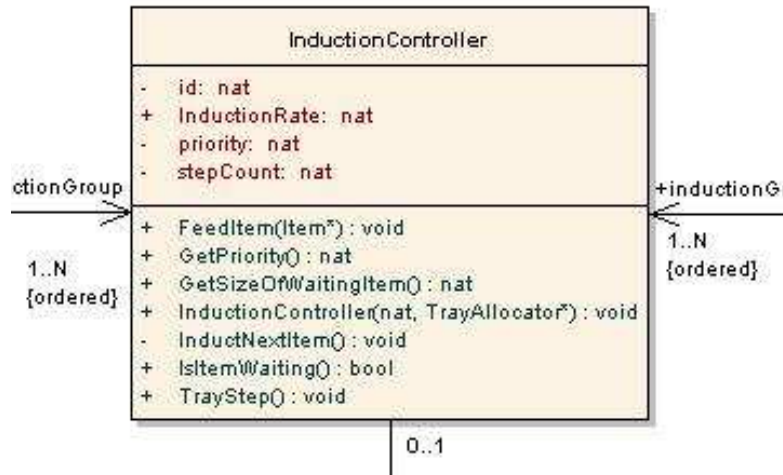


Figure 11: Detailed view of the induction controller class

The induction controller represents the device present at each induction. The induction controller is responsible for handling items received from the environment and handling the communication with the TrayAllocator class in order to induct items on the trays.

To reflect the association with the TrayAllocator class an instance variable has been declared (allocator). The association between items and induction controller needs a bit more elaboration. It should be kept in mind that the items arrive in order to the induction so the abstraction that should be used should allow the model to represent this order. A sequence in VDM is the most suitable abstraction in this case. When the system is initialized the value for this sequence of items is declared as an empty sequence.

```

class InductionController

  values
    public InductionRate : nat = 2;

  instance variables
    priority : nat := 0;
    id : nat1;
    allocator : TrayAllocator;
    items : seq of Item := [];
    inv priority > 0 => len items > 0;
  
```

In the case that an induction controller not being able to induct an item on the sorter ring at a given tray step the priority should be incremented by one. The item should be waiting at the induction and a retry will be performed in the next tray steps. This implies that the length of the sequence of items

must be higher than 0. This invariant is expressed just after the items declaration in the above model snippet.

Once the InductionController is constructed and the constructor invoked it is necessary to specify the allocator that is going to be associated with the Induction controller and the id that is assigned to it. This is done in the following lines of the model:

```
operations

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

The operation TrayStep simulates that the sorter ring has moved one tray step. At each tray step the induction controller tries to induct the next item (by invoking InductNextItem). It will try to induct the item in the case an item has been waiting (its priority will be higher than 0) or the number of counted steps is equal to the induction rate. The step counter is used to count the number of tray steps between inducted items allowing the model to maintain the specified InductionRate. In this sense inducing an item will automatically imply setting the value of stepCount to 0.

```
public TrayStep: () ==> ()
  TrayStep() ==
  (
    if IsItemWaiting() or (stepCount >= InductionRate)
    then
    (
      InductNextItem();
      stepCount := 0;
    )
  );
```

The operation InductNextItem will induct the first element contained in the sequence of items. The sequence will behave as a FIFO queue. The first element inserted in the sequence of items will be the first one to be extracted. The function checks if there are items contained in the sequence by checking if the length is greater than 0. It means that the first item in the sequence can be processed (done by the **hd** operator). The InductItem operation from the TrayAllocator is invoked using as parameters the current induction and the target item. In the case the item is successfully inducted on the sorter the sequence is updated that represents. This is done by assigning the tail of the items sequence to the items itself (with the **tl** operator all the elements contained at the sequence but the first will be considered). It is important to set the priority with value 0 after this assignation. Here it is needed to use the **atomic** operation otherwise the invariant concerning priority and length introduced at the beginning of this section will be broken. Considering no more items at the induction then the length of the sequence that contains the items will be 0 while the priority would be holding the value 2. As it

has been explained this situation is avoided by locating both operations in the atomic block alternative the order of the assignments should be changed.

```

private InductNextItem: () ==> ()
InductNextItem() ==
  let n = len items
  in
    if n > 0
    then
      let item = hd items
      in
        if allocator.InductItem(self, item)
        then
          atomic
          (
            items := tl items;
            priority := 0
          );
        else
          priority := priority + 1;
    end InductionController

```

5.2.3. The Tray Allocator class

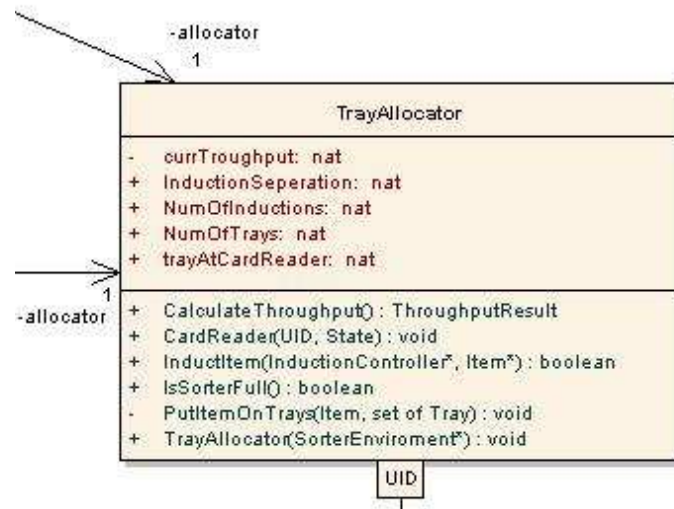


Figure 12: Detailed view of the Tray Allocator class

The TrayAllocator class is associated to a number of inductions. The number of inductions is defined in the values section of this class. The number of trays that are present in the sorter ring is also specified in this section. An invariant is defined to ensure that the number of trays should be greater than the number of inductions multiplied by the induction separation (measured in trays per induction). This invariant assures that the system has enough trays to compose a sorter. The InductionSeperation is the number of trays between each induction and the number of trays from the card reader to the first induction in the group. We assume in this model that this number is symmetric. (Same distance between all inductions and card reader to first induction in group)

```

class TrayAllocator

  values
    public NumOfInductions : nat = 3;
    public NumOfTrays : nat = 20;
    .....
  instance variables
    inv NumOfTrays > InductionSeperation * NumOfInductions;

```

In order to compute the throughput at a given configuration the TrayAlocator class is using two counters: the tray step counter and the items induced counter.

```

countTraySteps : nat := 0; -- Used for calculation of throughput
countItemsInduced : nat := 0;

```

As stated in requirement 11, "The system should be capable of working with several inductions per induction group" therefore the tray allocator must control several inductions defined in the induction group and the order should be taken in account in order to manage them properly. Again the abstraction that fits for this requirement is a sequence. It is applied by declaring a sequence of inductions that are contained in an induction group. An invariant can be used to specify that the number of inductions contained at the induction group sequence (obtained by the operator length) should be equal to the number of inductions in the system.

```

public inductionGroup : seq of InductionController := [];
inv len inductionGroup = NumOfInductions;
inv forall id in set inds inductionGroup
  & inductionGroup(id).GetId() = id;

```

The next invariant connected with the inductionGroup is using a quantifier to go through all the indexes contained in the sequence (of induction controllers) to ensure they will have the same id as the position in the induction group sequence.

In order to compose the sorterRing a relationship between the tray UID (Unique Identifier) and the trays must be created. After analysing the relation we have chosen to use and injective mapping. The **injective mapping** means that a tray UID should always map to only one tray. The mentioned feature is derived from the requirement 4 that states: "The system should assign a unique identifier per tray". This phenomenon is perfectly modelled by the **inmap** type which is a particular case of the map data type. The injective map connects elements from the domain to the range in creating a one to one mapping.

```

public sorterRing : inmap Tray`UID to Tray;
inv card dom sorterRing = NumOfTrays;
inv forall id in set dom sorterRing & sorterRing(id).GetId() = id;

```

The `sorterRing` is one of the main blocks of the system with an important role in order to analyze the model evolution. It is important to guarantee consistency on it by using invariants. In this way the cardinality of the domain will express the number of trays that compose the sorter. It should be equal to the public natural value declared at the beginning `NumOfTrays`. The domain of the map is a set of trays. The tray present at the card reader is connected with the UID contained in the variable `trayAtCardReader`. If the tray at the card reader value is greater than zero (Initialized to zero to specify unknown) then the detected tray must be contained in the set defined by the domain of the `sorterRing` map. The abstraction tray at card reader is connected with the requirement 12: "Each induction group should have a separate card reader"

```
public trayAtCardReader : Tray`UID := 0;
inv trayAtCardReader > 0 => trayAtCardReader in set dom sorterRing;
```

In order to allocate properly the items on the sorter several strategies have been considered. The strategy is distributed into the classes `AllocatorOneTray` and `AllocatorTwoTray`. An instance of each strategy is declared in the tray allocator class. Further details on the strategy pattern [8] and the implemented algorithms for each situation will be introduced in the section "The strategy pattern applied to the tray allocator"

```
oneTrayStrategy : AllocatorOneTray;
twoTrayStrategy : AllocatorTwoTray;
```

The tray allocator class constructor is responsible for initializing two key elements of the system: the sorter ring and the induction group. The sorter ring is initialized as follows. A value determined by num is mapped to a new Tray which is constructed using the value num as a parameter to the constructor. A set comprehension is used to create the `sorterRing` ensuring that all the trays that are mapped to a UID have the same id as the tray.

The induction group is created using a sequence comprehension creating a new induction controller class with parameters to the constructor self (`TrayAllocator`) and induction number (id) (in order to associate the tray allocator to the induction group).

```

operations

public TrayAllocator: SorterEnviroment==> TrayAllocator
TrayAllocator(e) ==
(
  sorterRing := {num |-> new Tray(num) |
                  num in set {1,...,NumOfTrays}};
  inductionGroup := [new InductionController(self, num) |
                     num in set {1,...,NumOfInductions}];
  .....
);

```

Once the sorter has moved one tray step the card reader associates a new tray UID in front of the card reader to the variable `trayAtCardReader`. This operation must satisfy the precondition that the UID must belong to the domain of the map that defines the sorter ring.

```

public CardReader: Tray`UID * Tray`State ==> ()
CardReader(uid, state) ==
(
  trayAtCardReader := uid;
  countTraySteps := countTraySteps + 1;
)
pre uid in set dom sorterRing;

```

The `InductItem` operation is responsible for inducting an item depending on its size measured in number of trays it occupies. In order to consider different parcel sizes specific implementations of the `AllocatorStrategy` class are used (the strategies previously declared as `oneTrayStrategy` and `twoTrayStrategy`). These strategies will be used depending on the result of the operation `GetSizeOfTrays` that belongs to the item object.

```

public InductItem: InductionController * Item ==> bool
InductItem(ic, item) ==
(
  dcl strategy : AllocatorStrategy;

  let numTrays = item.GetSizeOfTrays()
  in
    cases numTrays:
      1 -> strategy := oneTrayStrategy,
      2 -> strategy := twoTrayStrategy
    end;

  if strategy.InductionsWithHigherPriority(ic)
  then
    return false
  else
    let trays = strategy.AllocateTray(ic.GetId())
    in
      if trays = {}
      then
        return false
      else

```

```

        (
            countItemsInducted := countItemsInducted + 1;
            PutItemOnTrays(item, trays);
            return true;
        )
    )
    pre ic in set elems inductionGroup and item.GetSizeOfTrays() <= 2;

```

In case there exist one or more inductions with a higher priority than the one specified by *ic* then a false value will be returned and the induction will not be performed. If it is possible to induction the item on the sorter the strategy will be responsible for allocating one or two trays for the current induction id. The assigned trays will be returned as a set which can have one or two tray elements or the empty set (tray allocation was not possible) depending on the item size (connected to requirement 10). In this last case the operation will return false and the induction will not be performed. In the case the allocation has been performed successfully the counter of the inducted items is incremented and the item will be put on the tray or trays.

Putting an item on a tray is done by the operation *PutItemOnTrays*. Since it is possible that the item can have two trays assigned a set is used to model the relation. A loop statement is used to perform the operation *ItemOnTray* for all tray elements contained in the set.

```

private PutItemOnTrays: Item * set of Tray ==> ()
PutItemOnTrays(item, trays) ==
    for all t in set trays
    do
        t.ItemOnTray(item)
    pre trays <> {} and forall t in set trays & t.IsTrayEmpty();
end TrayAllocator

```

A precondition is defined to express that the set containing the assigned trays should be different from the empty set (an item can never be inducted in to the sorter if it has not got a tray associated). A set binding is used to express that assigned trays should be empty. In requirement 5: "Each tray must be occupied by one item at most" implies that the trays assigned to an item must be empty.

PutItemOnTrays could also be implemented as a recursive operation like in the following model snippet. Remark that conjunction in the precondition is now changed to an implication since the operation will be calling it self with an empty set.

```

private PutItemOnTrays: Item * set of Tray ==> ()
PutItemOnTrays(item, trays) ==
    if trays <> {} then
        let t in set trays
        in
        (
            t.ItemOnTray(item);
            PutItemOnTrays(item, trays \ {t});
        )
    pre trays <> {} => forall t in set trays & t.IsTrayEmpty();

```

5.2.4. The strategy pattern applied to the tray allocator

The strategy pattern [8] (also known as policy pattern) allows the model to swap dynamically the applied algorithm depending on the situation. This can be an advantage in several situations like the one involved in allocating items of different sizes on to the sorter. Depending on the item size different policies should be applied taking in consideration several conditions with different strategies. In the following class diagram is illustrated the logical structure of the strategy pattern applied to our problem. The TrayAllocator class represents the context of the strategy pattern. The AllocatorStrategy is the general strategy and the classes AllocatorOneTray and AllocatorTwoTrays are the concrete strategies.

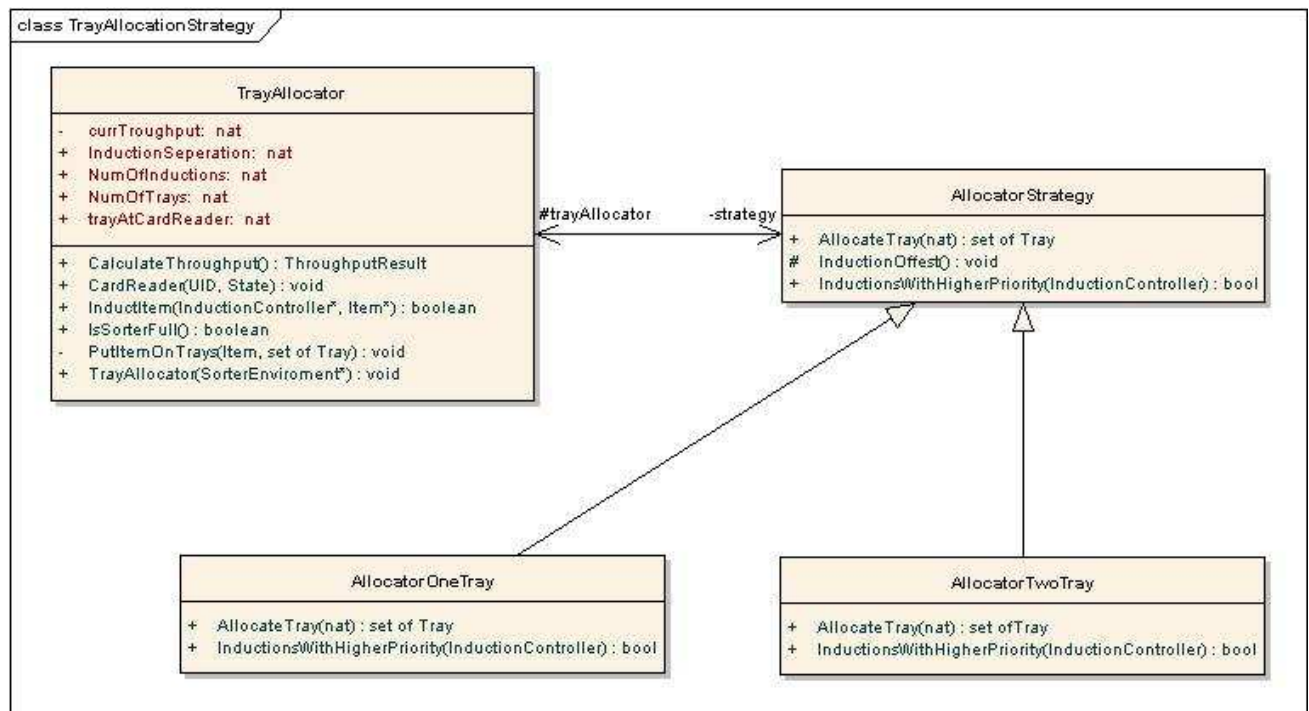


Figure 13: The strategy pattern in the Allocator Strategy context

An important advantage of this structure is that the AllocatorStrategy can easily be expanded with new algorithms. For example the system can be expanded to process three tray items or to process items using another priority policy reaching a higher level of maintainability and makes the model easy to extend.

The Allocator Strategy

The AllocatorStrategy class represents the general strategy that should be specialized in order to handle different situations.

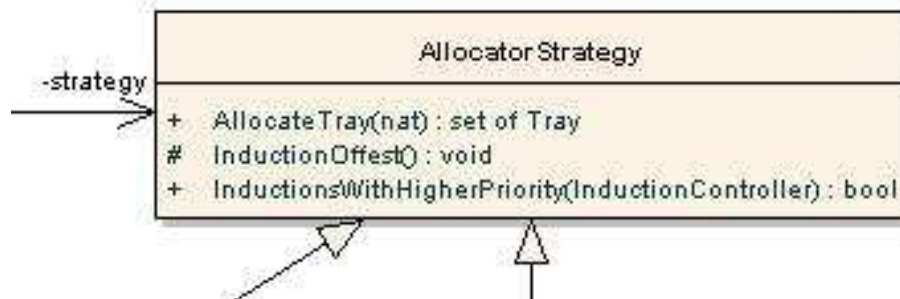


Figure 14: A detailed view on the Allocator Strategy

This class is using a protected variable called trayAllocator that models the association to the class TrayAllocator. By declaring this variable as protected the subclasses are allowed to access and read information from the TrayAllocator.

```

class AllocatorStrategy
    instance variables
        protected trayAllocator : [TrayAllocator] := nil;
  
```

The AllocatorStrategy class defines the public operations AllocateTray and are accessed by the trayAllocator class. Both methods have to be implemented at the concrete strategies therefore they are assigned with the statement **is subclass responsibility**.

```

operations

public AllocateTray: nat ==> set of Tray
AllocateTray (-) ==
    is subclass responsibility;

public InductionsWithHigherPriority: InductionController ==> bool
InductionsWithHigherPriority(ic) ==
    is subclass responsibility;
  
```

The InductionOffset function helps to compute the tray UID in front of the induction depending on the current tray that is detected at the card reader. This can be done because the trays are consecutively located and numbered by the UID. Considering the current detected tray and the relative distance from the induction to the card reader based upon induction number (id) and induction separation and adding the offset to the detected UID the tray in front of the induction can be calculated. No

preconditions are required since the calculation always returns a UID located at the sorter ring by using the mod operator.

```

functions

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

end AllocatorStrategy

```

The Allocator One Tray strategy

The AllocatorOneTray class is an example of how a concrete strategy is implemented. As its name express AllocatorOneTray will be responsible for associating a tray to the items that fit on a single tray.

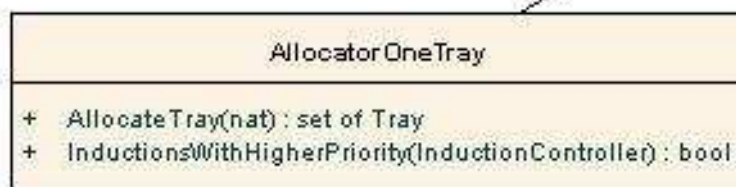


Figure 15: A detailed view on the Allocator One Tray strategy

Since this class is a subclass of the AllocatorStrategy the keywords **is subclass of** is added after class definition followed by the name of the super class.

```

class AllocatorOneTray is subclass of AllocatorStrategy

```

The AllocatorOneTray specifies the operations that have been defined as is subclass responsibility. The AllocateTray method will try to allocate the tray in front of the induction at the current tray step. A temporal variable posTray is defined to contain the UID of the tray in front of the induction. It is used to locate the tray at the sorter ring and determine if it is empty. This is an example of the advantage that resides on using a map to define the sorter ring. The tray can easily be found by its domain key in this case the UID.

In the case the tray is empty a set with one tray element is return. In case the tray is full an empty set is returned.

operations

```

public AllocateTray: nat ==> set of Tray
AllocateTray (icid) ==
  def posTray = InductionOffset(trayAllocator.trayAtCardReader, icid)
  in
    if trayAllocator.sorterRing(posTray).IsTrayEmpty()
    then return {trayAllocator.sorterRing(posTray)}
    else return {}
  pre icid in set inds trayAllocator.inductionGroup;

```

The operation AllocateTray should meet the precondition that states the induction id should refer to an induction that actually is contained in the indexes of the sequence used to model the induction group.

A situation to consider in the allocation algorithm is the possible existence of inductions with a higher priority waiting for an empty tray. This is implemented for the case of the one tray size item using the quantified expression exists. The elements contained in the sequence inductionGroup are the induction controllers. In the quantified expression listed below we are looking if there are other inductions in the group that has a higher priority than the current induction controller. In the case there is, the operation will return the Boolean value true, false otherwise.

```

public InductionsWithHigherPriority: InductionController ==> bool
InductionsWithHigherPriority(ic) ==
  return exists i in set elems
    trayAllocator.inductionGroup(1,...,
                                len trayAllocator.inductionGroup)
    & i.GetId() <> ic.GetId()
    and i.GetPriority() > ic.GetPriority()

```

This operation should satisfy a precondition that states the induction controller considered in this operation should belong to the induction group associated to the tray allocator.

```

  pre ic in set elems trayAllocator.inductionGroup;
end AllocatorOneTray

```

In first attempt to model the algorithm we were only looking for inductions in the group in front of the current induction controller, see model of InductionsWithHigherPriority below. During testing we found this method made starvation of the first induction in the induction group, since it would be waiting all time if the inductions in front of it were inducing items all time.

```

public InductionsWithHigherPriority: InductionController ==> bool
InductionsWithHigherPriority(ic) ==
  return exists i in set elems
    inductionGroup(ic.GetId()+1,...,len inductionGroup)
    & i.GetPriority() > ic.GetPriority()

```

6. Model system testing

This chapter describes how the 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 steps. 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 step number, 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.

Example for contents of scenario file:

```
mk_(4, [mk_(1,1,100),  
        mk_(1,2,200), - at time step 1, induction id 2, item size 200  
        mk_(1,3,900),  
        mk_(3,1,700), - at time step 3, induction id 1, item size 700  
        mk_(3,2,600)])
```

The ItemLoader class is added to the system and it handles the reading of the scenario files. The ItemLoader has operations to return the number of simulation time steps (GetNumTimeSteps) and returns a valid item size (>0) for a given time step and induction id (GetItemAtTimeStep). The ItemLoader is created by the World class and the SorterEnvironment which uses the ItemLoader during simulation to feed items to the inductions.

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

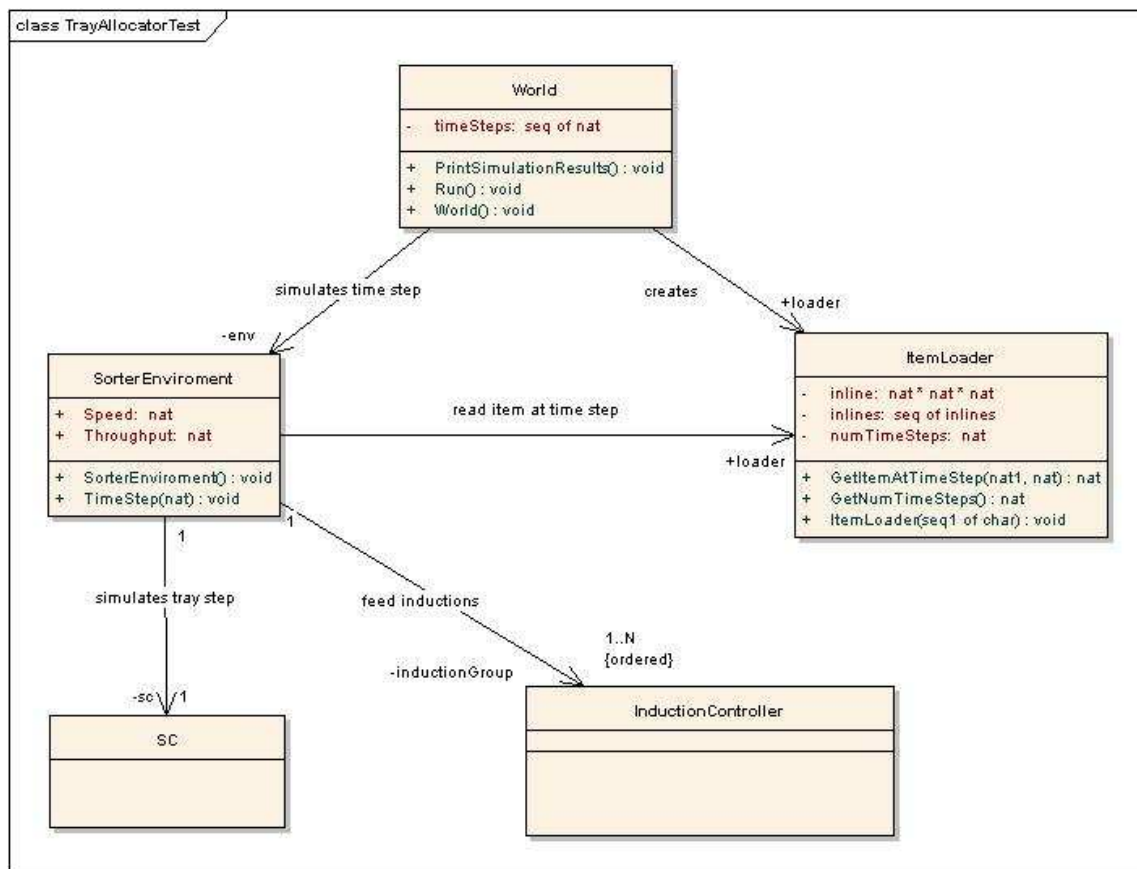


Figure 16: Class diagram for the environment including the ItemLoader

The World class Run() operation is described in the UML sequence diagram below. A sequence of test scenario files is specified and for every test scenario a complete new ItemLoader and SorterEnvironment is created. When the simulation is completed the operation PrintSimulationResults creates a report that is used to verify the result of the VDM model execution.

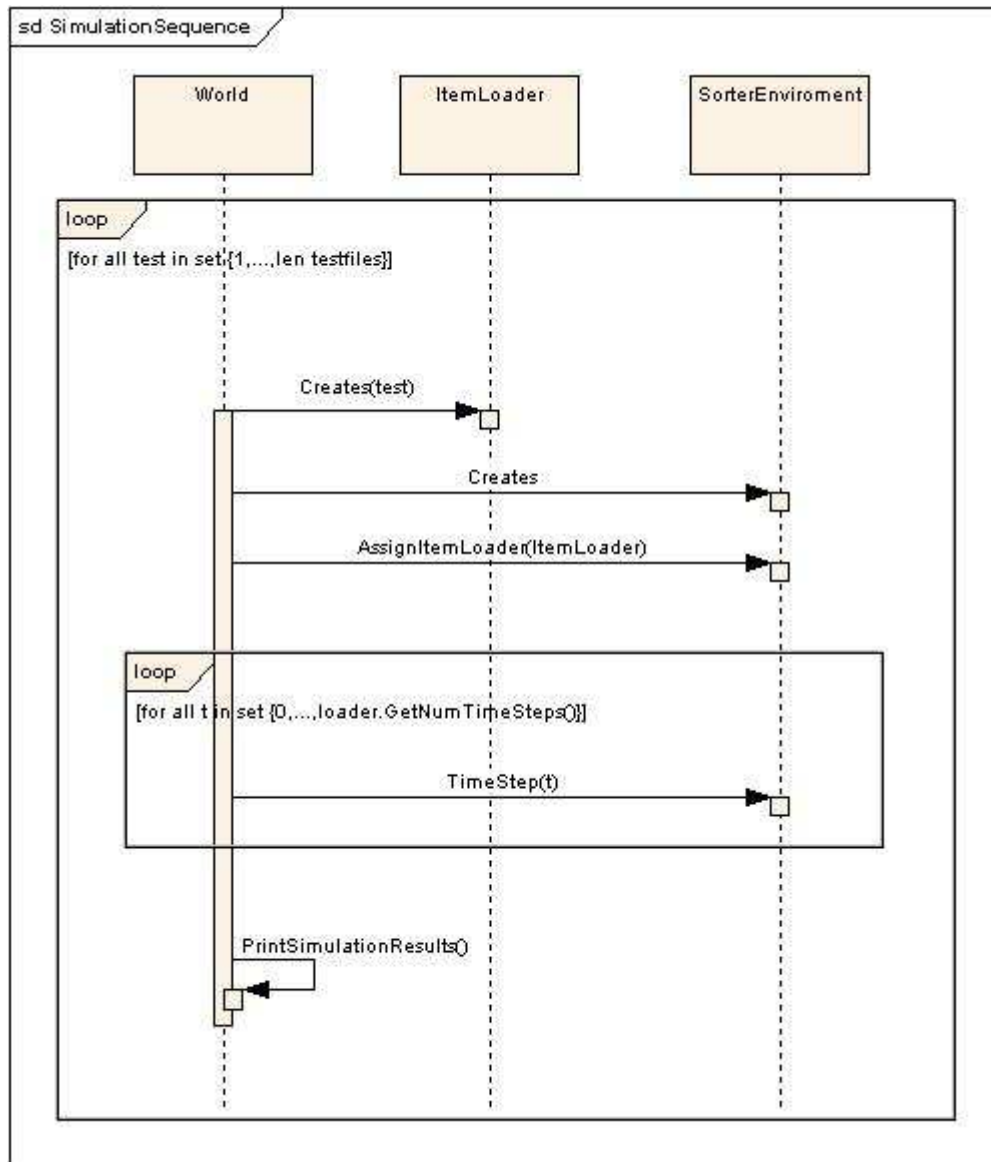


Figure 17: Sequence diagram for the Run operation in the Wold class

Below is listed the corresponding part of the VDM++ model that performs the **Run** operation of the **World** class creating a test scenario for every test file:

```
class World

  testfiles : seq of seq1 of char := [testfile1,
                                     testfile2,
                                     testfile3,
                                     testfile4,
                                     testfile5];

  public Run: () ==> ()
  Run() ==
  (
    for all test in set {1,...,len testfiles}
    do
      (
        env := new SorterEnviroment();
        loader := new ItemLoader(testfiles(test));
        env.AssignItemLoader(loader);

        IO`print("-----\n");
        IO`print("Tray allocation model test #" ^
                  String`NatToStr(test) ^ ": " ^
                  testfiles(test) ^ "\n");
        IO`print("-----\n");

        -- Performs simulation based on time steps
        for all t in set {0,...,loader.GetNumTimeSteps()}
        do
          env.TimeStep(t);

        PrintSimulationResult();
      );
    );
  );
```

When the simulation of time steps 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 model test #5 : scenario5.txt
-----
Card reader tray id 1
Card reader tray id 2
*Induction id 1
-> Item id 1 size 100 on tray id 5
*Induction id 2
-> Item id 2 size 200 on tray id 7
*Induction id 3
-> Item id 3 size 900 on tray id 9
-> Item id 3 size 900 on tray id 8
Card reader tray id 3
.....
Card reader tray id 7
*Induction id 1
-> Item id 4 size 200 on tray id 10
*Induction id 2
-> Item id 5 size 600 on tray id 12
-> Item id 5 size 600 on tray id 11
*Induction id 3
-> Item id 7 size 700 on tray id 14
-> Item id 7 size 700 on tray id 13
Card reader tray id 8
.....
*Induction id 1
-> Item id 8 size 600 on tray id 2
-> Item id 8 size 600 on tray id 1
*Induction id 2
-> Item id 12 size 800 on tray id 4
-> Item id 12 size 800 on tray id 3
*Induction id 3
-> Item id 13 size 300 on tray id 6
Card reader tray id 20
-----
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        : 3
Induction rate              : 2
Induction separation        [trays]: 2
-----
Number of trays with items      : 19
Two tray items on sorter       : 14
Number of tray steps           : 22
Number of inducted items       : 12
Calculated throughput[items/hour]: 10363
-----
      ****  Sorter is not full  ****
-----

```

In the example above the calculated throughput is above the specified throughput. We have in this example only one empty tray (Number of trays 20 – Number of trays with items 19). The simulation contains a mix of one and two tray items feeding the inductions during simulation. We can also see in the printed log that all inductions have been inducing items without starvation.

5 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 test case 1 and 2. When we have a mixture of one and two tray items, we still don't have starvation at any of the inductions, but the algorithm leaves empty trays on the sorter. Its performance is especially poor when the first induction in the group induces two tray items with the configuration we have used in the test scenarios.

1. One tray items only -> Calculated throughput[items/hour]: 10909
2. Two tray items only -> Calculated throughput[items/hour]: 10909
3. Mix of one/two tray items -> Calculated throughput[items/hour]: 9272 (3 empty trays)
4. Mix of one/two tray items -> Calculated throughput[items/hour]: 8727 (4 empty trays)
5. Mix of one/two tray items -> Calculated throughput[items/hour]: 10363 (1 empty tray)

The simulation result is calculated by calling the operation CalculateThroughput in the TrayAllocator class see VDM++ model below. This operation creates a record ThroughputResult that is updated based on the status of the trays on the sorter ring, counted tray steps and counted items induced.

```

class TrayAllocator
  types
    public ThroughputResult::
      traysWithItemOnSorter : nat
      twoTrayItemsOnSorter : nat
      traySteps : nat
      inducedItems : nat
      calcThroughput : real;

  operations
  public GetThroughput: () ==> ThroughputResult
  GetThroughput () ==
    CalculateThroughput(countTraySteps, rng sorterRing,
                        countItemsInducted);

  functions
  private CalculateThroughput: nat * set of Tray * nat-> ThroughputResult
  CalculateThroughput(steps, trays, items) ==
    let runTime :real = steps * (Tray`TraySize/SorterEnviroment`Speed),
        traysWithItems = {twi | twi in set trays & twi.IsTrayFull()},
        traysWith2Items = {tw2i | tw2i in set traysWithItems
                             & tw2i.GetItem() <> nil
                             and tw2i.GetItem().GetSizeOfTrays() = 2},
        itemsOnSorter = card traysWithItems,
        twoTrayItemsOnSorter = card traysWith2Items,
        throughput = itemsOnSorter * SecInHour/runTime

    in
      mk_ThroughputResult(itemsOnSorter, twoTrayItemsOnSorter,
                          steps, items, throughput)

    pre trays <> {};

end TrayAllocator

```

During the testing we have found and fixed a number of errors in the model especially. Invariants and preconditions have been especially helpful while finding these errors. The model has been tested with different tray allocation strategies, like changing the strategy for inducing 2 tray items if only other inductions within the same group are also waiting for 2 tray items.

The debugger interactive console has been used to extract a sorter ring status view after each test scenarios have stopped at a breakpoint in the function CalculateThroughput. In the example below a set comprehension is used to make a print of all trays containing an item, by creating a set of tuples containing the tray id, item id and size.

Test scenario 2 – 2 tray items:

```
=>{mk_(t.GetId(), t.GetItem().GetId(),t.GetItem().GetSize()) | t in set rng
sorterRing & t.IsTrayFull()}
"{mk_(1, 9, 900), mk_(2, 10, 600), mk_(3, 10, 600), mk_(4, 1, 600), mk_(5,
1, 600), mk_(6, 2, 700), mk_(7, 2, 700), mk_(8, 3, 800), mk_(9, 3, 800),
mk_(10, 4, 900), mk_(11, 4, 900), mk_(12, 5, 600), mk_(13, 5, 600), mk_(14,
6, 700), mk_(15, 6, 700), mk_(16, 7, 700), mk_(17, 7, 700), mk_(18, 8,
800), mk_(19, 8, 800), mk_(20, 9, 900)}"
```

Test scenario 3 – mix of 1 and 2 tray items:

```
=>{mk_(t.GetId(), t.GetItem().GetId(),t.GetItem().GetSize()) | t in set rng
sorterRing & t.IsTrayFull()}
"{mk_(1, 11, 800), mk_(2, 11, 800), mk_(4, 12, 200), mk_(5, 1, 100), mk_(6,
2, 800), mk_(7, 2, 800), mk_(8, 4, 200), mk_(9, 3, 200), mk_(10, 5, 400),
mk_(11, 6, 700), mk_(12, 6, 700), mk_(13, 7, 800), mk_(14, 7, 800), mk_(16,
8, 300), mk_(18, 9, 400), mk_(19, 10, 900), mk_(20, 10, 900)}"
```

- Prints empty trays:

```
=>{t.GetId() | t in set rng sorterRing & t.IsTrayEmpty()}
"{3, 15, 17}"
```

- Prints priority of inductions:

```
=>{ mk_(i.GetId(), i.GetPriority()) | i in set elems inductionGroup}
"{mk_(1, 4), mk_(2, 4), mk_(3, 4)}"
```

Besides the model system testing we have generated Proof Obligations for the complete VDM model and checked for possible runtime errors, deadlocks and state invariants. In the below figure a proof obligation error is indicated for picking the head of the sequence of items in the induction controller, but we see that a check for the number of items is performed before "hd items".

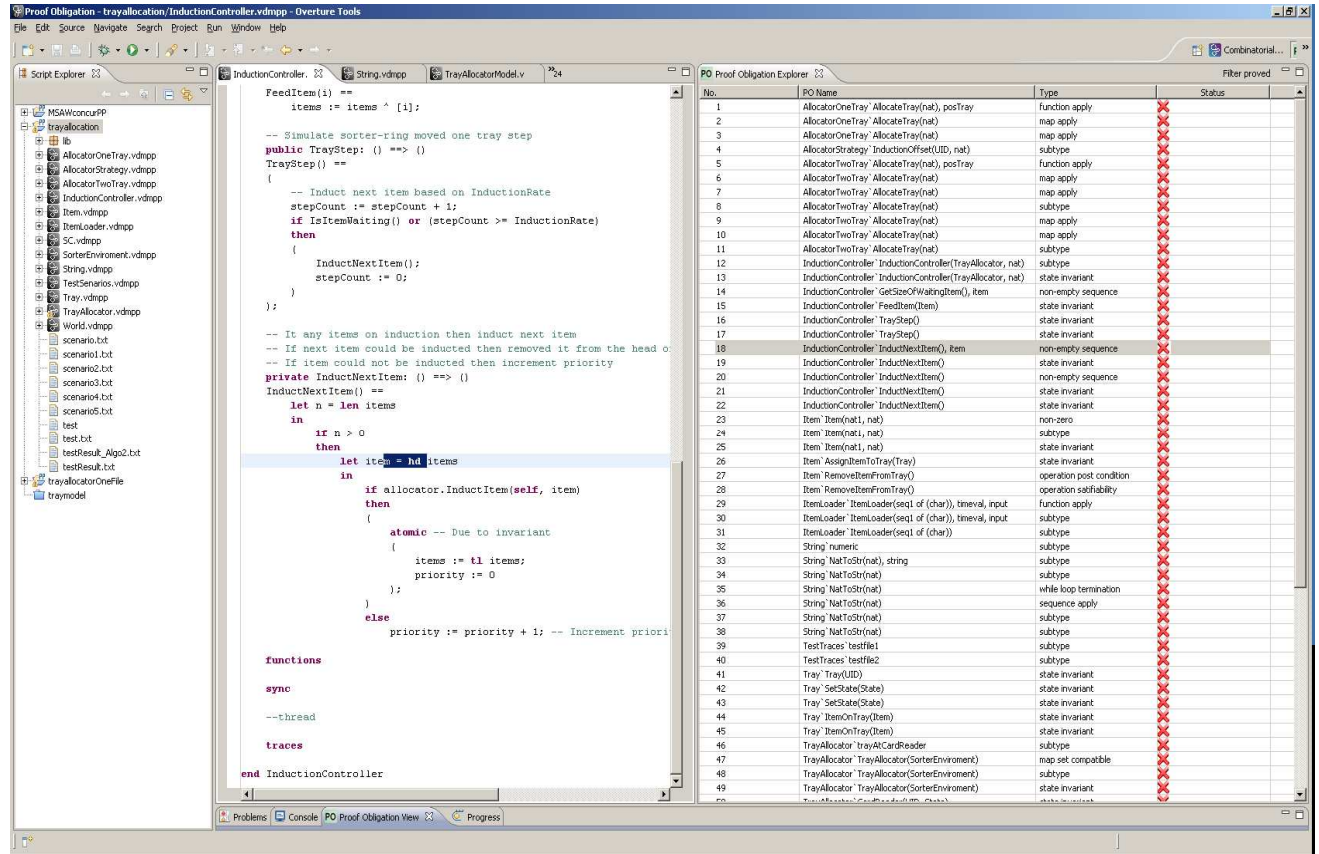


Figure 18: Proof Obligation of the VDM model in overture

Finally we have tried to perform a fully automated test of the whole model with traces, but did not succeed since reading the scenario files from traces did not seem to work as expected. The TestScenario trace below is supposed to run all the combinations of test files performing all combinations of time steps.

```
class TestTraces
instance variables
    env : SorterEnviroment := new SorterEnviroment();
    testfile1 : seq1 of char := "\\scenario1.txt";
    loader1 : ItemLoader := new ItemLoader(testfile1);
    testfile2 : seq1 of char := "\\scenario2.txt";
    loader2 : ItemLoader := new ItemLoader(testfile2);
    tests : set of ItemLoader := {loader1, loader2};

traces
TestScenario: (
    let loader in set tests
    in
    (
        env.AssignItemLoader(loader);
        let step in set
            {1,...,loader.GetNumTimeSteps()}
        in
            env.TimeStep(step)
    )
);
end TestTraces
```

7. Conclusion

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.

We have found that valuable information can be extracted without requiring specific hardware or complex software by just using modelling and **abstractions**. These abstractions has been useful because they were defined with accuracy by using a language that allows us to be precise expressing the functionality in terms of relations (Ordered, unordered and mappings), entities (records, tuples and classes), methods (operations and functions), attributes (types, values and variables) and restrictions (invariants, pre and post conditions).

An important task in the project has been to create good **requirement** as a fundament for a precise model with the right abstractions. This is a vital issue that determines if the model that has been developed actually is valid fulfilling the requirements presented for the problem. We have kept in mind these requirements through the whole modelling and documentation process of the project. As a result of this activity we can assure that the created model is actually a reflection of the real problem domain. These requirements has been explained and referenced constantly while the different classes have been described in the previous sections.

As part of the learning objectives of the course we have covered many of the VDM modelling concepts and data abstractions presented in the lectures, and use the different modelling elements that have been introduced during the whole quarter. In this project we have covered many modelling aspects like **defining data and functionality** (realized in classes, data types, values, variables, operations and functions), **modelling unordered collections** by using sets (Use in associations between classes in the UML diagrams), **modelling ordered collections** by using sequences (for instance, applied to define the inductions and its position at the induction group), and **modelling relationships** by applying mappings (this abstraction have been used to represent the sorter and the injective relation between a UID and a tray).

Seen from a practical point of view, in the case that software should be developed to control the sorter in real life the next step would be to use the VDM++ model as a golden reference from where the implementation of the VDM++ model could be generated and integrated with the target system. This integration will be possible due to the **separation between the environment and the logic** controller classes. In this way several components could be easily communicating with the controller classes in order to allow communication with sensors and actuators (for instance signals coming from the card reader, photo sensors detecting items and motor controllers that handles the control of item insertion to the sorter from the inductions). A remarkable feature of this executable model

specification is that it has been validated and tested before starting the real implementation. This approach ensures that the specification is valid and logical problems and errors have been detected very early in the development before the final implementation and testing phases.

8. References

- [1] Fitzgerald, Larsen et Al. "Validated Designs for Object-oriented Systems". Springer 2005
- [2] Larsen, Peter Gorm. Slides and materials from the course "Model driven development using VDM++ and UML 1". Aarhus Engineering College. Spring 2010.
"http://kurser.iha.dk/eit/tivdm1/materiale.html" [Link valid march 2010]
- [3] "The Crossorter":. Multimedia resource available at youtube showing a sorter ring in action.
"http://www.youtube.com/watch?v=8FILmtVHCAQ&feature=related" [Link valid march 2010]
- [4] VanderLande Industries: official website of a sorter system manufacturer, illustrating several applications and types of sortation systems. "http://www.vanderlande.com/web/Parcel-Postal/Sortation/Loop-sorters/CROSSORTER.htm" [Link valid march 2010]
- [5] "Crossorter for Taxipost": Multimedia resource available at VanderLande Industries official youtube channel, showing a concrete application of a sorter in the parcel handling at taxipost (Belgian Post). "http://www.youtube.com/watch?v=Zf9zyRkGfDM" [Link valid march 2010].
- [6] "Formal methods": Wikipedia article on formal methods. General information and references.
http://en.wikipedia.org/wiki/Formal_Methods [Link valid march 2010]
- [7] "Viena Development Method": Wikipedia article on VDM. General information, code examples and references. http://en.wikipedia.org/wiki/Vienna_Development_Method [Link valid march 2010].
- [8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides , "Design Patterns Elements of Resusable Object-Oriented Software" Addison Wesley, pages 315 – 324
- [9] "Overture – Open-source Tools for Formal Modelling"
http://www.overturetool.org/twiki/bin/view [Link valid march 2010]
- [10] "Crisplant Automated Post and Parcel Sortation Systems"
http://www.crisplant.com/en/Logistic-Systems/Post-and-Parcel.aspx