Modelling of an agent based control system for a model factory with the specification language Chi

Eindhoven
University of Technology
The Netherlands

Research Report

FACULTY OF TECHNOLOGY MANAGEMENT

Modelling of an agent based control system for a model factory with the specification language χ

by
Arian Zwegers, Raymond Schrijver,
Angel Santana Alguacil

Report EUT/BDK/88
ISBN 90-3860-569-2
ISSN 0929-8479
Eindhoven 1997

Modelling of an agent based control system for a model factory with the specification language χ

by

Arian Zwegers, Raymond Schrijver, Angel Santana Alguacil

Report EUT/BDK/88
ISBN 90-3860-569-2
ISSN 0929-8479
Eindhoven 1997

Keywords: Agent based control systems / Manufacturing systems / Simulation

Eindhoven University of Technology
Faculty of Technology Management
Eindhoven, The Netherlands

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Zwegers, Arian

Modelling of an agent based control systems for a model factory with the specification language x / by Arian Zwegers, Raymond Schrijver, Angel Santana Alguacil. - Eindhoven : Technische Universiteit Eindhoven, 1997. -
(Report EUT/BDK, Eindhoven University of Technology, Department of Industrial Engineering and Management Science, ISSN 0929-8479 ; 88)
ISBN 90-3860-569-2
NUGI 684
Subject headings: Agent based control systems / Manufacturing systems / Simulation

Summary

This report describes the specification and simulation of an agent based control system for a model factory. A manufacturing system can be regarded as a set of autonomous, problem solving *agents*, which communicate with each other. Each agent is capable of executing one or more tasks. By means of negotiation, agents see to it that operations are carried out.

The real-time, concurrent programming formalism χ is used to specify and simulate the model factory. The formalism χ treats a manufacturing system as a set of simultaneously operating sequential components. Interaction among components is modelled by send- and receive-actions along fixed communication lines.

The model factory is a miniaturised model of Printed Circuit Board assembly and test plant. It consecutively carries out the following operations: releasing empty boards, screen printing, placing components, reflow & cleaning, and storage. Each workstation is accompanied by an agent. The workstation agents are connected by a network through which the exchange of messages takes place. Jobs are passive entities that flow through the system; they do not have agent capabilities and therefore they cannot negotiate. The workstation agents negotiate with each other about the execution of jobs.

The control system is based on a push-approach. A job is taken into the system via the Raw Material Store. Subsequently, the job seeks its way through the system. The goal of a workstation agent is to negotiate about the process steps of a job, so that the job will be completed. The workstation agents announce an operation. Workstations that are able to execute the operation respond with a bid that contains the point of time at which the operation could be completed. The job is allocated to the workstation agent that is able to finish the operation first. A workstation has the opportunity to subcontract operations, i.e. to carry out part of the process steps itself, and have the other part carried out by another workstation.

The model is kept as generic as possible, for instance by means of the communication network between the workstation agents. Moreover, new workstations can be added to the system without modification of the rest of the model. As a result of this genericity, the structure of a workstation agent is more complicated than strictly necessary.

Experiments have been done with an agent based control system and a similar control system without negotiation. As for performance, the first system performs slightly better than the latter. However, the performance and robustness of the agent based control system compared to other control forms depend mostly on the type and characteristics of the production system. Furthermore, an agent based control system appears to require a lot of communication. Especially, the possibility of subcontracting enlarges the number of messages over the network considerably.

The formalism χ imposes hardly any constraints on the implementation of the models. The basic structure of χ is quite clear. However, this does not guarantee the transparency of the model.

Table of Contents

List of Figures

List of Tables

1. Introduction

Current production management architectures show significant deficiencies in controlling the complexity and the uncertainty that is typical of manufacturing systems. In manufacturing systems, the predominant architectural paradigm has up to now been hierarchical. Because of its mechanistic and deterministic approach, the hierarchical paradigm has numerous defects in coping with uncertainty and with the rapidly evolving scenario that characterises today's manufacturing environments. In this report, an approach is adopted that is derived from Distributed Artificial Intelligence, and that is based on the concept of distributed, autonomous agents.

This chapter discusses the principles behind agent based control systems. Furthermore, the specification language χ is described. The object of the studies in this report, the model factory, is presented, and the objective of this report is outlined.

In Chapter 2, the architecture of the agent based control system is outlined. The next chapter elaborates on the architecture and specifies the control system in χ. Chapter 4 gives the simulation results of the χ model, thereby verifying the specified control system. This report concludes with a discussion of the control system and the specification language χ.

1.1 Agent Based Control Systems

Strong similarities can be found between the characteristics of agents and those of current manufacturing systems. Manufacturing processes are highly dynamic and unpredictable; it is difficult to completely separate the planning and sequencing of required activities from their execution. Any detailed time plans are often disrupted by unpredictable delays and other unanticipated events. As a result, a tendency exists within manufacturing systems to decentralise the ownership of the tasks, information, and resources involved in the various processes. Different groups within manufacturing systems become relatively autonomous; how their resources are consumed, by whom, at what cost, and in which time frame lies within their own prerogative.

Given these characteristics, it is quite natural to model the processes in a manufacturing system as a collection of autonomous, problem solving agents which interact when they have interdependencies. In such a context, an agent can be seen as an encapsulated problem solving entity that exhibits the following properties:

- *Autonomy*: agents perform the majority of their problem solving tasks without the direct intervention of other agents; they control their own actions and their own internal state.
- *Social ability*: agents interact, when they deem appropriate, with other agents in order to complete their problem solving and to help others with their tasks. This implies that agents have, as a minimum, a means by which they can communicate their requirements to others and an internal mechanism to decide what and when social interactions are appropriate (both in terms of generating requests and judging incoming requests).
- *Proactiveness*: agents take the initiative where appropriate.
- *Responsiveness*: agents perceive their environment and respond in a timely fashion to changes that occur in it (Jennings *et al.*, 1996).

Each agent is able to perform one or more services or tasks. If an agent requires a service that is managed by another agent, it cannot simply instruct the other agent to start the service; agents are

autonomous, and control dependencies between them do not exist. Instead, the agents must come to a mutually acceptable agreement about the terms and conditions under which the desired service will be performed. The mechanism for making these agreements is negotiation, a joint decision making process in which the parties verbalise their demands and then move towards agreement by a process of concession.

To negotiate with one another, agents need a protocol that specifies the role of the current message interchange, e.g. whether the agent is making a proposal or responding with a counterproposal, or whether it is accepting or rejecting a proposal. A well-known example of such a protocol is the Contract Net (Smith, 1980). According to this protocol, agents decide upon their actions by exchanging demand and offer for services among themselves, together with varying amounts of status information which depend on the selected implementation approach. However, Van Brussel (1995) notices that this protocol is often called a negotiation procedure, but that there is no real negotiation involved. The protocol only defines a set of rules that state how allocations are to be made.

1.2 Specification Language χ

At Eindhoven University of Technology, department of Mechanical Engineering, a real-time concurrent programming formalism has been developed, called χ. This formalism can be used for the specification and simulation of industrial systems. It supports modularity and allows separate descriptions of the structure and of the components' behaviour. A specific feature of χ is the clear representation and unambiguous specification of interfaces between components (Mortel-Fronczak *et al.*, 1995).

A system is treated as a collection of concurrently operating sequential components. A system component is modelled by a process as a sequential program where changes in the state of a process are accomplished by performing actions. Interaction between components is modelled by send and receive actions along fixed communication channels. A process is specified by a program in a CSP-like specification language preceded by Pascal-like declarations of local variables and statistical distributions. Processes do not share variables – they interact exclusively by using the communication and synchronisation primitives (synchronous message-passing). The reader may find an extensive example of the specification language χ in (Mortel-Fronczak *et al.*, 1995; Chi, 1996).

1.3 Model Factory

The model factory is a miniaturised, though still complex, model of a real Printed Circuit Board (PCB) assembly and test plant. The function of the model factory is to assemble and test pseudo PCBs. The following subsections describe the product, operations, and process lay-out of the factory.

1.3.1 Product

The model factory produces printed circuit boards. Each PCB consists of a board and a maximum of six components. Currently, two different types of boards and three types of components are used in the model factory.

1.3.2 Operations

The model factory emulates operations which are performed on real PCBs during the manufacturing process. The operations of the model factory have been derived from case studies of real PCB manufacturing facilities. These operations are:

- *screen printing*: the bare PCB is positioned in the workstation, a PCB-specific screen is selected and moved into position, and a squeegee is reciprocated horizontally over the screen.
- *component placement*: the PCB is positioned in the workstation, and components are placed on the positions according to the component-placement recipes for that product.
- *reflow and cleaning*: PCBs are passed through an oven and cleaning station
- *test and repair*: the PCB is inspected to see if it contains the components in the designated position, and component and functional tests are performed. If the PCB fails, it must be routed to an off-line diagnosis and repair workstation. Upon successful repair, the PCB is routed back to the test station.

1.3.3 Process Layout

In addition to the operations described above, the model factory contains some other features. Raw material and components are automatically supplied from a centralised raw material store and component store respectively. The model factory can support mixed model flow production, where different types of products can be manufactured at the same time. The model factory is designed for batch production, but the batch size can vary from batch to batch, as well as product to product. The maximum batch size in the model factory is three.

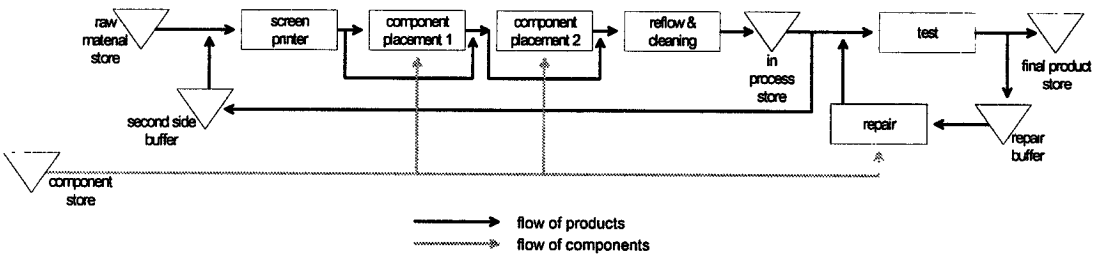


Figure 1 Primary process of the model factory

The process layout is depicted in Figure 1. The operations are indicated by square boxes, whereas stock points are indicated by triangles. The first stock point contains the two different empty board types. All products pass the screen printer, but alternative routings are possible between the two component placement stations. After the reflow and cleaning station, the batches may be stored in the in-process-store which consists of three locations for three products each. Here, a batch can be split or concatenated with other batches. Then, products are tested and – if necessary – repaired. In the test and repair cycle, a maximum of one batch can reside in the buffer. Finally, nine individual products can be stored in the finished-product-store.

An additional feature is a loop from the in-process-store to the screen printer. This loop is necessary to manufacture PCBs that have components on both sides. These products have to pass the process twice, since only one side can be finished in one pass. The buffer in this second-side loop may contain only one batch (Timmermans, 1993a; Timmermans, 1993b).

1.4 Problem statement

Shop floor control architectures are studied within the department of Technology Management at the Eindhoven University of Technology in the Netherlands. The model factory is owned by the department, and is used as a test site for implementation and evaluation of various control architectures.

The objective of this report is to present the design of an agent based control system for the model factory, and to compare the characteristics of the agent based architecture with those of a previously specified control architecture. The architecture is specified by means of χ, since this formalism is suitable for the specification of distributed control architectures. The agent based design is subsequently compared to another heterarchical control system, which was previously designed and implemented in the model factory.

1.5 Evaluation criteria

The following criteria are used to evaluate the agent based control system:

- performance
- flexibility
- robustness

The performance concerns the (average) completion time of various samples of a large number of jobs. The (structural) flexibility concerns the 'ease' with which the control system can be extended or modified, if a new workstation is added or changed. Robustness concerns the ability to deal with disturbances. For the moment, the robustness of the control system has not been evaluated by means of the χ model.

2. Global Modelling

In this chapter, the architecture of the agent based control system is outlined. The reasons for choosing a push approach are explained, and the negotiation model and its accompanying protocols are described. Finally, the modelling of a workstation and the transportation system are briefly discussed.

2.1 Introduction

Although the model factory is a scale model, and the operations are fake, the control system is still quite complex. In order to simplify the models, a few assumptions are adopted. Note that these assumptions do not influence a comparison between the performances of the agent based control system and other types of control systems:

- The production system is not subject to defects. The only breakdown the system could handle is that of the component placers, but only one at the same time. In that case, all operations could still be performed.
- The production system is capable of executing the operations as stated in the recipe for a product type. In other words, the production system is able to produce the desired product.
- The production system receives orders from some planning system. A job order states how the product should be manufactured. Issue of the board is the first step. The next step is screen printing. Then, the recipe states which components have to be placed at the top side of the board. Subsequently, reflow & cleaning takes place. The next step is either transport of the board to the final product store or to screen printing for the second side. If the second side is operated upon, the recipe indicates which components have to be placed. Also in the second side loop, the board goes to the reflow & cleaning station, after which it goes to the final product store.

Furthermore, the model factory has a few constraints:

- A workstation does not have an output buffer. The space between two workstations is considered as the input buffer for the last workstation. The absence of an output buffer implies that the boards processed by a workstation are moved to the input buffer of the next workstation. This means that the next workstation has to be known before a workstation starts processing a batch. Furthermore, the assumption is made that multiple batches may reside in an input buffer; the workstation is intelligent enough to make a distinction.
- There are only two kinds of product carriers, i.e. two types of boards, and three types of components. Only the difference in the type of components is taken into account; the difference in type of boards is disregarded.
- The workstations are capable of executing only one type of operation. Placing green, red, or yellow components is one type of operation. Component placement may vary in the number and position of the components to be placed.

2.2 System Boundaries

Figure 2 presents the system to be modelled. It is the part of the model factory from the Raw Material Store to the Final Product Store, the loop for second side printed circuit boards, and the component delivery subline. The In-Process-Store is not used as such, and is therefore disregarded, just like the Test & Repair loop.

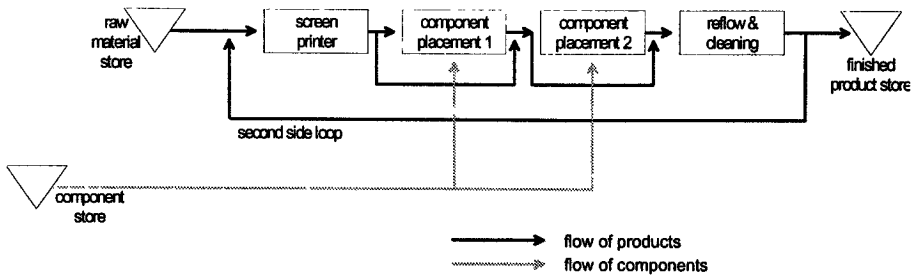


Figure 2 Part of the model factory to be modelled

Every workstation consists of a part that is responsible for communication and negotiation on behalf of the workstation. Figure 3 shows that agents are connected by means of a network that is used for message exchange. Compared to direct channels between every pair of workstations, the network significantly reduces the number of channels (Coenen, 1995).

The workstations negotiate among each other about the execution of the jobs. Batches/jobs are passive entities flowing through the system. They do not have agent-like capabilities, and therefore are incapable of negotiation. This choice is made in particular because χ assumes fixed communication channels, so components such as job agents can not be created and deleted, but have to exist permanently. However, since the number of jobs simultaneously being operated upon cannot exceed a certain maximum, some tricks might get around this restriction (see e.g. (Coenen, 1995)).

The interface between the information flow and the material flow takes place via the physical parts of the workstations. They receive and send messages from/to the machine controllers and they receive and send batches from/to the conveyors. The physical parts of the workstations are interconnected by means of conveyors. These conveyors do not have the capabilities of an agent.

2.3 Control Strategy

2.3.1 Push Strategy

Batches can be pushed through or pulled out of the factory. With a pull-approach, a planning is made in advance. The last station in the line, in casu the Final Product Store, is requested to deliver a batch of finished products at a certain due date. Then, the last station requests the appropriate batch from its preceding station, which – at its turn – asks for semi-finished batches to its predecessors, and so on. When a complete planning is made, the order is released and production starts. For an example of such a system, the reader is referred to (Wiendahl and Ahrens, 1995).

In this report, an opportunistic push-approach is chosen. With a push-strategy, the job is brought into the system at the first point of the line, namely the raw material store.

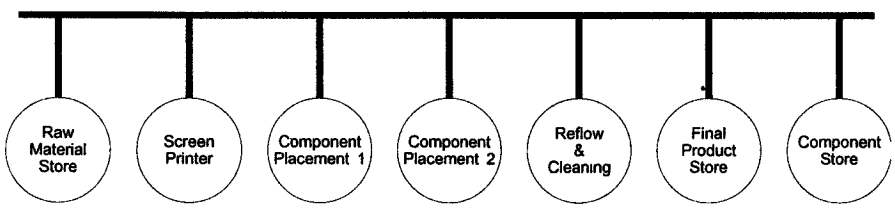


Figure 3 Agents connected by a network

Subsequently, the job finds its way through the system. However, in the model factory a convergent material flow is present at the component placement stations; both the boards and the components lead to these assembly stations. In general, since operations are not planned before job dispatch, stock points should be created in order to decouple the main stream from the branches. These buffers can be replenished by means of a pull approach and simple inventory control heuristics. Just in front of the model factory's component placement stations, small buffers are located in which two component trays, each containing four components, are stored. If the first tray in a buffer is out of components, a new tray is ordered from the central component store. Upon arrival of the new tray with components at the buffer, the empty tray is removed. In the χ model, however, only one buffer for component trays is modelled. New components are ordered when the number of present components is not sufficient to fulfil a job.

The main advantage of an opportunistic dispatch method is that decisions concerning the distribution of work on the shop floor are based on the prevailing system status rather than on some projection of that status (as would be the case with a pull approach). Disadvantages include the fact that opportunistic schedulers are myopic, that they may ignore interactions with other components, and that they may only handle priorities in a rather cumbersome way (Upton *et al.*, 1991). Advantages and disadvantages of both approaches are displayed in Table I.

Table I Advantages and disadvantages of the push and pull approaches

	Advantages	Disadvantages
Push	• Robust, capable of dealing with disturbances • Opportunistic behaviour, routing flexibility	• Less suitable for convergent material flows • Only short-term vision, possibly myopic
Pull	• Suitable for convergent material flows • Plans in the future	• Excessive planning needed, possibly myopic • More sensitive to disturbances

2.3.2 Negotiation

The agent based control system is based on negotiations between entities, in casu the workstations, in order to coordinate the activities of the physical manufacturing system. The workstations are supplied with agents that represent the workstations in the negotiation model. The objective of an agent is to negotiate about the operations of a job in order to execute these operations. The workstation agents offer other agents tasks to perform an operation consisting of one or more process steps. Workstations that are capable of execution of the operation reply with a bid that contains the point of time at which the operation could be finished. The workstation agent that could finish the process step first is awarded with the job.

2.3.3 Negotiation protocol

As stated above, workstation agents negotiate with each other. A protocol is needed to coordinate and control these negotiations. The point of departure is that agents only have information at their disposal about the workstations they represent; they have no information about other workstations. A workstation agent has access to the following data that is relevant to the negotiation model:

- the process steps a workstation has to perform; this information is passed from one agent to the other as a batch flows from one workstation to the other;

- the physical possibilities of the workstation, namely the operations a workstation is able to perform, and the operation times;
- the components available to the workstation, and their replenishment times;
- the up to date schedule for the workstation.

The workstation agent that is about to execute an operation searches a workstation for execution of the next operation. After all, a workstation does not have an output buffer, and it has to put processed boards in the input buffer of the next workstation. All operations together form the job. An operation might consist of multiple process steps that are all of the same type of operation. A workstation might execute all process steps in an operation, or it might distribute execution of the process steps among itself and another workstation. This is called '*subcontracting*', and is obviously only possible if an operation contains multiple process steps.

A workstation agent sends a task announcement for the next operation of the job that is about to be executed by the workstation. This task announcement is sent to all workstation agents connected to the network, a so-called '*broadcast*'. Note that a broadcast to all workstation agents is needed since an agent does not have any information about other workstations in the system. A workstation agent that receives a task announcement only replies with a bid if the operation can be performed at that workstation. The message saying that a workstation cannot perform an operation would only cause more communication via the network, and is not sent. After a certain period of time, the agent that sent the task announcement chooses the best bid that has been received up to that moment. The time limit is needed because a workstation agent cannot know how many agents will react on a task announcement. Another option would be to choose the agent that has sent the first incoming bid. It is clear that this might cause a suboptimal overall system performance. Due to the assumption that the production system is capable of executing the process steps of a job, an agent that sent a task announcement will always receive at least one bid. After a bid has been selected, a task offer is sent to the workstation agent with the best bid.

However, subcontracting causes some exceptional situations. A workstation agent expects task announcements to perform operations. An announcement is replied with a bid if the workstation is able to perform the operation. If the operation consists of multiple process steps, the agent sends a subcontracting task announcement by means of a broadcast throughout the network. The subcontracting task announcement is meant for all combinations of half or less of the total number of process steps to be contracted. A main contractor never performs less process steps than a subcontractor. A possible subcontractor cannot try to subcontract the subcontracted process steps again. The agent receives one or more subcontracting bids. The workstation agent combines the best subcontract bid (to be executed by another agent) with the accompanying main contract (to be executed by the agent itself). The agent compares this bid containing subcontracting with the bid where it performs all process steps itself. The best bid is sent to the workstation agent that broadcasted the task announcement. Task offers are only sent to main contractors. So, if a task offer is received for a bid with subcontracting, the main contractor notifies the subcontractor with a subcontracting task offer.

Summarised the protocol is as follows:

1. Before a workstation agent A performs an operation on a job, it checks whether the workstation for the next operation is already determined due to subcontracting.
   a. In case the next operation has not been fixed yet, agent A broadcasts a task announcement for the next operation to all other agents.
      If a workstation agent B is able to perform the (next) operation, it might do the following actions, depending on the number of process steps in the operation.

If the operation contains multiple process steps, agent B broadcasts a subcontracting task announcement.

Other agents that are able to perform the subcontracting process steps reply with a subcontracting bid.

Agent B selects the agent with the best subcontracting bid, and drafts a bid with subcontracting that might be sent to agent A.

Agent B chooses the best possible bid (with or without subcontracting) and sends it to agent A.

After a certain period is expired, agent A determines which agent will perform the next operation, and sends a task offer to that agent.

b. In case the next process steps are already determined by subcontracting, no special actions are taken.

2. Agent A starts its own operation.
3. When agent A is finished with its operation, it sends the batch and accompanying job information to the next workstation.

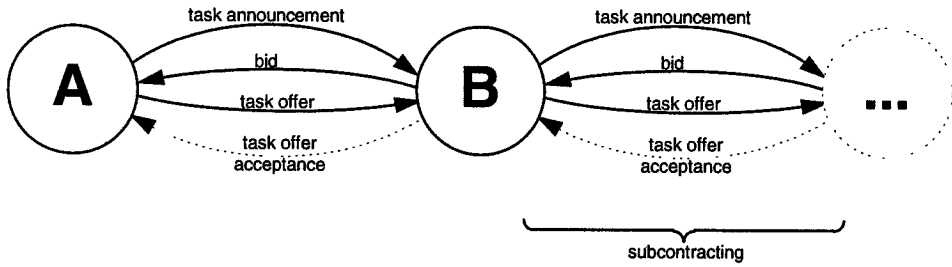


Figure 4 Negotiation protocol

Figure 4 shows the negotiation protocol applied by the workstation agents. Compared to the original version, the Contract Net Protocol (Smith, 1980), the task offer acceptance is omitted and subcontracting is added.

2.4 Workstations

Workstations perform operations of one or more process steps on batches. A batch consists of at most three boards; the maximum batch size is three. There is one, generic model of a workstation. The only differences among workstations are their capabilities, their operation times, possible component supply and its duration, and their place in the manufacturing system represented by the physical part of the workstation. The processing of an operation is modelled by waiting a certain time period.

2.5 Transportation System

A transportation system connects the workstations to each other. This system consists of conveyor belts that allow multiple batches to be simultaneously transported. The transportation duration from one workstation to another is proportional to the distance between the two workstations, i.e. proportional to the absolute difference between the two workstation identification numbers. Conveyor belts (except the component conveyors) do not allow queuing of batches; only one batch can be present at the same time.

2.6 Some Specific Problems and Their Solutions

Due to the peculiarities of the model factory and the negotiation protocol, some difficulties might occur for which a solution has to be designed. In this section, some of these problematical situations and their specific solutions are discussed. Concrete examples are given for the specific configuration of the model factory.

Situation 1: jobs overtaking older jobs

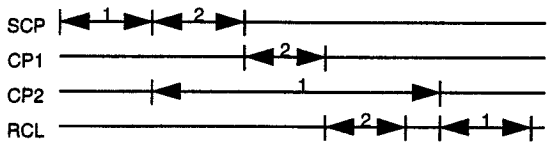


Figure 5 Jobs overtaking older jobs

In Figure 5, job 2 overtakes job 1 at the Reflow & Cleaning station. Before CP1 starts processing, it broadcasts a task announcement. With the job information in the task announcement (process steps, batch size, and first possible start time), RCL should be able to schedule this job before the earlier planned job 1. A function *PLANNING* in the Database makes this possible.

Situation 2: jobs waiting for processing capacity

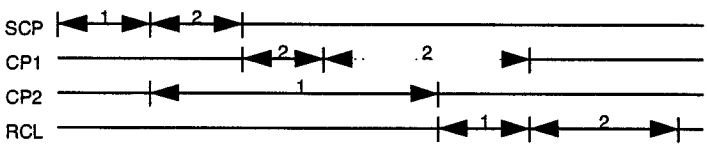


Figure 6 Jobs waiting for processing capacity

In Figure 6, job 2 has to wait in workstation CP1 until RCL has finished processing job 1, and is ready to receive job 2. After all, CP1 does not have an output buffer, and it is therefore not able to process a next job. The dashed line represents a waiting job. Before CP1 starts processing, this station broadcasts a task announcement. RCL checks its planning and 'provisionally plans' job 2 after job 1. Note that jobs are only (firm) planned when task offers are received. RCL issues a bid in which it states the planned end time for job 2. This is the criterion for CP1 to select the next workstation. However, besides the planned end time, RCL should also return the planned start time. CP1 will not select a workstation with this data, but it will update its own schedule with it (corrected for transportation times). After all, job 2 will stay in CP1 until RCL can receive it. Jobs are sent from one station to the other on the basis of the (updated) planning.

An additional measure would be to 'physically lock' stations when a batch is present, i.e. the incoming physical channels of a workstation, as represented in Figure 20, would not be ready to receive a new batch if a batch is already present. However, this measure would not suffice because of the second side loop (see situation 4).

Another additional feature would be to make planning more intelligent. For example, at the moment RCL receives a task announcement for job 2, it has already planned job 1. In the

current situation, it will plan job 2 after job 1. If it had more intelligence, it would plan job 2 before job 1, so that job 1 has to wait in CP2. Overall performance would be improved, but RCL has to send a message to CP2 saying that job 1 has to wait. However, a choice has been made for fixed schedules, unless jobs have priority (see situation 4).

Situation 3: obsolete schedules

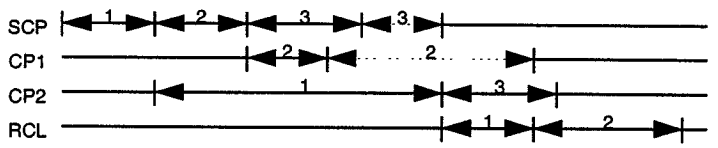


Figure 7 Obsolete schedules

In Figure 7, a problematic situation is outlined at the moment SCP received job 3. At that moment, SCP broadcasts a task announcement for job 3. CP1 has to offer a bid for that task announcement. At the same moment, CP1 broadcasts a task announcement for job 2. Before CP1 can send a valid bid to SCP for job 3, it has to receive a bid from RCL for job 2, and it has to update its schedule based on the accepted bid of RCL. Because of the absence of queues, batches have to reside in a workstation until the next workstation is ready to receive them. Therefore, it will frequently occur that two batches move from one workstation to the other at the same time. The model cannot guarantee that a workstation (here: CP1) receives bids, selects the best bid, and updates its schedule before it itself sends a bid.

A possible solution for this problem is as follows. When the task announcement for job 3 arrives, CP1 knows it is about to broadcast (or has just broadcasted) a task announcement for job 2. After all, CP1's database has recorded that a new batch should arrive at the same moment. Therefore, CP1 issues a bid with reservation. The Screen Printer will receive two bids: one with reservation from CP1 and one from CP2. SCP waits until CP1 sends its definite bid, i.e. a bid without reservation. In the meantime, SCP allegedly starts processing the batch, since the processing time of a board is much larger than the time needed to determine the next workstation. In the χ model, SCP starts processing when it sends a task offer; during the negotiation process, the physical part of the workstation is idle.

However, another solution has been chosen: instead of sending bids with reservation, a workstation sends an *invalid bid.* In Figure 7, CP1 issues an invalid bid for job 3. The Screen Printer will receive two bids: an valid one from CP2 and an invalid one from CP1. SCP knows its task announcement procedure was not correct, and it waits a small period of time. Then, it tries the whole task announcement procedure again, hoping that it will receive only valid bids this time. Indeed, if CP1 has updated its schedule while SCP was waiting to send the task announcement for job 3 for the second time, SCP will receive valid bids.

Situation 4: obsolete schedules and the second side loop

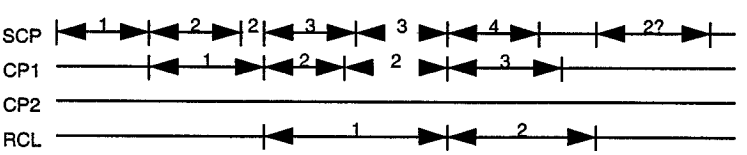


Figure 8 Obsolete schedules and the second side loop

In Figure 8, a similar situation as in Figure 7 is given, but CP2 is left out of consideration for the sake of discussion. Job 2 has to return to the Screen Printer via the second side loop. In this situation, all workstations are waiting for each other. When RCL receives job 2, it broadcasts a task announcement for this job. At the same time, SCP and CP1 broadcast task announcements for job 4 and 3 respectively. All three stations have to send bids on the basis of incomplete schedules. With the solution as outlined above, the three stations would be waiting for *valid* bids for ever. Note that if job 2 would not return to the Screen Printer via the second side loop, RCL would receive a valid bid (from the Final Product Store), and there would be no problem.

The best solution would be as follows. It is clear that the bottleneck, or rather the 'largest' job, i.e. the most time consuming job, determines the moment at which the whole system could perform its next 'rotation'. In this case, the largest job is job 2 at RCL. When job 2 would be finished, RCL could receive job 3, CP1 could receive job 4, and SCP could receive job 2. A hierarchical controller overlooking all stations could easily determine the bottleneck and inform the workstations about it. However, this would violate the agents principle. Another solution would be to have the individual workstations determine such a situation by themselves, and have them determine the bottleneck. This would call for a (very) complicated solution and a lot of message exchange between agents to find the solution.

Therefore, the following solution is chosen: *priority is given to the second side loop.* This means that if a station (here: SCP) receives a task announcement from a station farther down the line (here: RCL), it gives priority to this job. SCP 'preliminarily plans' job 2 as indicated in Figure 8, and sends a valid bid to RCL. This allows RCL to send a valid bid to CP1, and so on.

This solution would work optimally if Reflow & Cleaning would be the bottleneck. However, consider the situation in Figure 9 where job 3 at CP1 is the bottleneck. Based on the algorithms as described above, the Screen Printer would receive job 2 while job 4 would still be present and waiting for CP1. Therefore, when SCP receives a bid from CP1 and updates its planning for job 4, it will notice a collision with the earlier planned job 2. If stations would be 'physically locked' when batches were present, there would be no problem. However, then also batches could not be physically sent to another station for the same deadlock reason. Another solution would be to assume that Reflow & Cleaning is always the bottleneck. However, especially due to large varieties in batch sizes, this is most unrealistic.

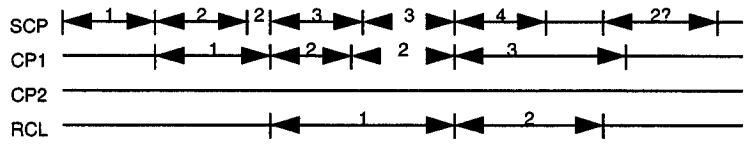


Figure 9 Another bottleneck

The solution chosen is to have SCP inform RCL that job 2 has to wait at RCL until SCP can send job 4 to CP1. This would make RCL update its planning; possibly, RCL would have to inform other workstations about its updated planning, and so forth. It is a 'message-intensive' solution, but the only right one.

3. Detailed Modelling

In this chapter, the architecture as outlined in the previous chapter, is elaborated upon and specified in the formalism χ. The χ code is given with minimal error detection provisions and output messages.

3.1 Naming

The names of the various processes are either written with a capital for the first letter, or are abbreviated to three capitals. The names of the communication channels are composed of the three-letter names of the sending process, followed by the names of the receiving process, written in lowercase. If a process has multiple channels with the same name, then the channels are indexed. Data types start with a capital 'T', followed by a name that indicates the type. Lists start with a lowercase 'x'.

3.2 Data Types

The data types that are used in the specification of the agent based control architecture for the model factory are enumerated below:

- The agents in the production system are uniquely identified. The type of this identification is:

  Tid = nat

  The agents communicate with each other by sending messages via a network. The address of an agent is identical to its identification. A message sent to address 0 has to be sent to all agents (broadcast), except to the sender itself. Workstations are identified as in Table II.

Table II Unique identification of workstations

Workstation	Identification
(broadcast)	0
Generator (GEN)	1
Raw Material Store (RMS)	2
Screen Printer (SCP)	3
Component Placement 1 (CP1)	4
Component Placement 2 (CP2)	5
Reflow & Cleaning (RCL)	6
Final Product Store (FPS)	7
Component Store (COS)	8

- Agents negotiate with each other via a network. Negotiations take place by means of message exchange. The data in a message concerns amongst others the address of origin, the target address, and the type of message. The address of origin is the agent that sends the message. The target address is the identification of the agent to which the

message has to be sent. A workstation needs to know the type of message in order to react adequately. Instead, different channels for different message types could be used, which would result in a tremendous amount of channels between workstations. The data type of the message type is:

Ttype = nat

All message types are given in Table III.

Table III Message types

Nr	Message type	From an agent's	To other agents'
1	task announcement	Sender	Request Handler
2	bid without subcontracting	Request Handler	Sender
3	bid with subcontracting	Request Handler	Sender
4	task offer without subcontracting	Sender	Request Handler
5	task offer with subcontracting	Sender	Subcontractor
6	subcontracting task announcement	Subcontractor	Request Handler
7	subcontracting bid	Request Handler	Subcontractor
8	subcontracting task offer	Subcontractor	Request Handler
9	job information	Controller	Controller
10	component order	Database	—
11	planning change	Database	Database
12	invalid bid without subcontracting	Request Handler	Sender
13	invalid bid with subcontracting	Request Handler	Sender
14	invalid subcontracting bid	Request Handler	Subcontractor
15	finished job	Controller	—

- Workstations negotiate with each other about the process steps that have to be carried out. At a certain moment in time, messages about a certain job are only about one operation of that job; it is not possible to negotiate about a certain operation, while messages about previous operations of the same job are still being sent via the network. Note that at the same time, messages may be sent about different process steps of one and only one operation. Each job is uniquely identified by a job number and an identification for possible subcontracting. These data types are respectively:

  Tjob = int

  Tsubjob = int

- The next variable in Tmessage is used for the first agent in the line that should receive the physical batch: either the main contractor or the subcontractor. An agent that sends a task announcement does not need to know whether bids contain subcontracting or not. However, it needs to know to which workstation it has to send the batch, when the agent is finished with its job. Similarly, in case a main contractor sends subcontracting task announcements, it needs to inform possible subcontractors if the batch will come from the main contractor or from the agent that sent the original task announcement. Therefore, **Tid** is used for the (first) agent that should receive / send the batch.

- Besides a job number, a job consists of a batch size and process steps. Batches contain one, two, or three products. The type of the batch size is:

  Tbatchsize = int

  A job consists of multiple lines that each represent one process step. Every line contains the sequence number of the process step, the type of operation that has to be carried out, and the number of the workstation that should perform the operation. This last attribute is not used in the agent based control system, but it is used in an experiment with a control system without negotiation (see next chapter). Note that there might be several process steps with the same sequence number, e.g. for placing yellow and green components. The type of a process step is:

  Tprocstep = < **Tseqnr # Top # Tinfo** > = < int # int # int >

  All possible operations, *Top*, are listed in Table IV. The last column shows the workstations that are able to perform the operation types (see also Table V).

Table IV Possible operations

Top	Operation type	Workstations
1	board dispatching	RMS
2	screen printing	SCP
3	placing yellow components	CP1, CP2
4	placing red components	CP1, CP2
5	placing green components	CP1, CP2
6	reflow & cleaning	RCL

- A message might contain zero, one, or two points of time. For a task announcement, a message contains the moment at which the current operation is finished, and the next operation could start. For a bid, a message contains the points of time at which the execution of the operation will start and will be finished. The latter is the criterion on which the selection of bids is based. For a task offer, both time points are given. In other messages, such as job information messages (type 9), no time points are given.

  Tvalue = real

- The type of (network) messages is:

  Tmessage = < Tid # Tid # Ttype # Tjob # Tsubjob # Tid # Tbatchsize #
  <Tseqnr # Top # Tinfo>* # Tvalue # Tvalue>

  The first *Tid* is the address of origin, the second is the target address, and the third is the address of either the previous or next agent. The third *Tid* is used to indicate the workstation from which a batch will be received or to which a batch will be sent.

- Components are identified at the same manner as their corresponding placement operations. The components 3, 4, and 5 are related to the operations 3, 4, and 5.

- A workstation has information about its own capabilities, operation times and the delivery duration of its components, if applicable. These are recorded in a list of records with the following type:

Topdata = <Top # real # real>

Table V Operation times and component delivery duration (xop)

Work-station	Type of operation (text)	Type of operation (number)	Variable operation time per board	Component delivery duration
RMS	board dispatching	1	10	-
SCP	screen printing	2	20	-
CP1	placing yellow components	3	20	5
	placing red components	4	20	5
	placing green components	5	20	5
CP2	placing yellow components	3	20	5
	placing red components	4	20	5
	placing green components	5	20	5
RCL	reflow & cleaning	6	20	-
FPS	final product store	7	60	-

- In order to negotiate about subcontracting, sets of process steps with the same sequence number are split, which results in tasks to be carried out by the main contractor and the subcontractor. The type of a task is as follows:

  Ttask = <Tjob # Tsubjob # Tprocstep* # Tprocstep* # Tvalue # Tvalue>
  = <Tjob # Tsubjob # < Tseqnr # Top # Tinfo >* # < Tseqnr # Top # Tinfo>* # Tvalue # Tvalue >

All types discussed above and remaining types are:

- Tid = nat
- Ttype = nat
- Tjob = int
- Tsubjob = int
- Tbatchsize = int
- Tseqnr = int
- Top = int
- Tinfo = int
- Tprocstep = <Tseqnr # Top # Tinfo>
- Tvalue = real
- Tmessage = <Tid # Tid # Ttype # Tjob # Tsubjob # Tid # Tbatchsize # <Tseqnr # Top # Tinfo>* # Tvalue # Tvalue>
- Tplan = <Tjob # real # real # real # Tid # Tbatchsize # < Tseqnr # Top # Tinfo>*>,
- Tcompstock = <int # int>
- Ttask = <Tjob # Tsubjob # Tprocstep* # Tprocstep* # Tvalue # Tvalue>
- Topdata = <Top # real # real>
- Trequest = <Tjob # Tid # Tbatchsize # < Tseqnr # Top # Tinfo>* # Tvalue>
- Treply = <Tjob # Tvalue # Tvalue # Tid>
- Tdestin = Tid
- Tbatch = Tjob
- Ttaskoff = Tmessage

3.3 Generator (GEN)

Function

The function of the Generator is to send jobs into the system. Jobs are either retrieved from a data file or automatically generated. The Generator connects with the workstation agents via the Switch Element. There is little communication between the Generator and the agents. Because of this, and due to the behaviour of the Generator, it does not need a Network Interface, which acts as a buffer between the Switch Element and agents. However, a design decision is made to connect the Generator, the Component Store, and all workstation agents to the network by means of Network Interfaces (see Section 3.4).

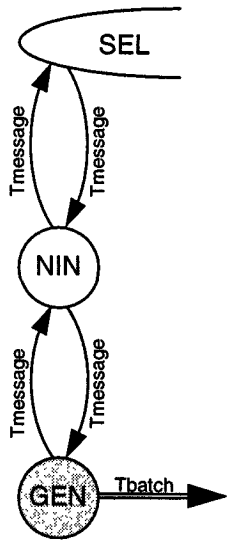


Figure 10 Generator

Interface

gennin : task announcement, or task offer to the Raw Material Store
ningen : bid from the Raw Material Store to the Generator
genmacrms: physical batch from the Generator to the physical part of the Raw Material Store

Behaviour

(1) Firstly, the Generator is initialised. Jobs might either be read from a data file or are generated automatically. The total number of jobs is requested from the user or the data file. As long as the total number of jobs has not been sent into the system, the Generator will be generating.

(2) If a message is received from the Switch Element, the Generator performs one of the following actions:

- In case a bid with or without subcontracting (*m.2 = 2 or m.2 = 3*) is received, the time that determines the moment at which the next job is sent is updated with the first possible start time minus the distance between the Generator and the next workstation, the Raw Material Store (*m.8 - 5*). The Generator sends a task offer to the agent that sent the bid, the Raw Material Store. Note that the Generator knows its next station is the Raw Material Store. All other stations do not have this kind of knowledge. Obviously, this could easily be changed by equipping the Generator with Sender-like capabilities. The variable *sendnextjob* becomes true; the Generator is able to send the job since a task offer has been sent.
- In case a planning change message (*m.2 = 11*) is received, the variable *timersend* is updated with the value of the first possible start time (*m.8*). Note that in planning change messages,

the first possible start time is updated for the transportation time. In fact, *m.8* is not the first possible start time for the next agent, but the first possible 'finish time' for the Generator.

- In case the message is an invalid (subcontracting) bid (*m.2 = 12 or m.2 = 13*), the Generator makes the necessary preparations to send the previous task announcement again.
- In case the message is a job finish message (*m.2 = 15*), the variable *nrinsystem* is decreased. If the maximum number of jobs in the system was reached and thus the variable *waiting* was set, the variables *waiting*, *generatenextjob*, and *timergenerate* are reset. A new job will be generated as soon as possible (*timergenerate := time*).

(3) If a job is allowed to enter the system (*sendnextjob* is true and *delta (timersend - time)* equals zero), the Generator sends the job information and the batch (*nr*) to the Raw Material Store. The job sequence number and the total number of batches in the system are updated. In case the number of jobs in the system is lower than the maximum number that is allowed, the next job can be immediately generated (*generatenextjob := true; timergenerate := time*). If the system is maximally loaded, the next job cannot be generated yet, and the Generator waits for a 'finished job' message (*waiting := true*).

(4) If the next job is allowed to be generated (*generatenextjob* is true, the maximum number of generated jobs has not been reached yet, and *delta (timergenerate - time)* equals zero), the Generator either reads the next job from a data file or generates it automatically, depending on the variable *data*. Note that in case jobs are automatically generated, the probability of having a second side loop is determined by the variable *probsec*. After the job is read or generated, the Generator sets the parameters to send the new task announcement.

(5) If the next task announcement may be generated (*sendnewtaskann* is true and *delta (sendnewttasktimer - time)* equals zero), the Generator sends a task announcement to the Raw Material Store

Code

```
proc GEN (id: Tid, t: real, maxnr: nat,
          gensel: ! Tmessage, selgen: ? Tmessage, genmacrms: ! Tbatch, lambda: real) =
|[ ps: Tprocstep,                -- process step
   xps: Tprocstep*,              -- list of process steps
   total, nr2,                   -- total number of components, total second loop
   bs: int,                      -- batch size
   nrjobs,                       -- nr of jobs
   seq, op, info, nr,            -- sequence nr, operation, info, nr
   batchesfinished,              -- number of batches finished
   nrinsystem: int,              -- number of jobs in the system
   data: nat,                    -- data file (1) or automatic generation (2) of jobs
   d :-> int,                    -- distribution of a random number
   f :-> real,                   -- probability distribution second side loop
   prob2: real,                  -- sample probability second side loop
   iat:-> real,                  -- inter arrival time distribution
   sendnewtasktimer,             -- point of time at which a task announcemnt should be sent
   timersend,                    -- point of time at which the next job should be sent
   timergenerate: real,          -- point of time at which the next job should be generated
   probsec: real,                -- probability second side loop
   sendnewtaskann,               -- should a task announcement be sent?
   sendnextjob: bool,            -- is next job ready to be sent?
   generatenextjob: bool,        -- can the next job be generated?
   waiting: bool,                -- GEN is waiting for a job to be finished (type 15)
                                 -- in order to generate the next job
   m: Tmessage                   -- message
 | nr := 1; nrinsystem := 0; iat := nex(lambda); probsec := 0.20;                    (1)
   timergenerate := 10; sendnextjob := false; generatenextjob := true;
   sendnewtaskann := false; waiting := false; batchesfinished := 0;
   ! "Maximum number of jobs in the system ", maxnr,nl();
   ! "Jobs in data file (1) or automatically generated (2)?? "; ? data;
   ! "Number of jobs = "; ? nrjobs; ! nl();
   *[ selgen ? m ->                                                                  (2)
              [ m.2 = 2 or m.2 = 3 ->
                     timersend := m.8 - 5;
                     gensel ! <id, 2, 4, nr, 0, id, bs, [hd(xps)], m.8, m.9>;
                     sendnextjob := true;
              | m.2 = 11 ->
                     timersend := m.8
```

```
            | m.2 = 12 or m.2 = 13 ->
                  sendnewtaskann := true;
                  sendnewtasktimer := time + 2 * t;
            | m.2 = 15 ->
                  nrinsystem := nrinsystem - 1;
                  !time, " GEN, Job ",m.3," is finished!!",nl();
                  batchesfinished := batchesfinished +1;
                  !time, " GEN, ",batchesfinished," batches are finished",nl();
                  [ waiting ->
                        waiting := false;
                        generatenextjob := true; timergenerate := time
                  | not waiting ->
                        skip
                  ]
            ]
      | sendnextjob; delta (timersend - time) ->                                    (3)
            sendnextjob := false;
            gensel ! <id, 2, 9, nr, 0, 0, bs, xps, 0, 0>;
            genmacrms ! nr;
            nr := nr + 1;
            nrinsystem := nrinsystem + 1;
            [ nrinsystem <= maxnr - 1 ->
                  generatenextjob := true; timergenerate := time
            | nrinsystem = maxnr ->
                  generatenextjob := false; waiting := true
            ]
      | generatenextjob and nr <= nrjobs; delta (timergenerate - time) ->           (4)
            generatenextjob := false;
            xps := []; op := 0;
            [ data = 1 ->
                  ? bs;
                  *[ op /= 7 ->
                        ? seq; ? op; ? info;
                        ps := <seq, op, info>;
                        xps := xps ++ [ps]
                  ];
            | data /= 1 ->
                  d := dun (1,4); bs := sample d;
                  xps := xps ++ [<1,1,0>]; xps := xps ++ [<2,2,0>];
                  d := dun (1,7); total := sample d;
                  f := cun (0,1); prob2 := sample f;
                  [ prob2 < probsec and total >= 2 ->
                        d := dun (1,total); nr2 := sample d;
                  | prob2 >= probsec or total < 2 ->
                        nr2 := 0
                  ];
                  info := 1;
                  *[ total - nr2 > 0 ->
                        d := dun (3,6); op := sample d;
                        ps := <3, op, info>; xps := xps ++ [ps];
                        info := info + 1; total := total - 1 ;
                  ];
                  xps := xps ++ [<4,6,0>];
                  [ nr2 = 0 ->
                        xps := xps ++ [<5,7,0>];
                  | nr2 /= 0 ->
                        xps := xps ++ [<5,2,0>];
                        *[ nr2 > 0 ->
                              d := dun (3,6); op := sample d;
                              ps := <6, op, info>; xps := xps ++ [ps];
                              info := info + 1; nr2 := nr2 - 1 ;
                        ];
                        xps := xps ++ [<7,6,0>];
                        xps := xps ++ [<8,7,0>]
                  ]
            ];
            sendnewtaskann := true; sendnewtasktimer := time;
      | sendnewtaskann; delta (sendnewtasktimer - time) ->                          (5)
            sendnewtaskann := false;
            gensel ! <id, 2, 1, nr, 0, id, bs, [hd(xps)], time, 0>;
      ]
   ]
]|
```

3.4 Network

The network connects the workstation agents, the Generator, and the Component Store. Via a Network Interface, these processes are connected to the Switch Element of the network. Although a Network Interface is a generic part of a workstation agent, i.e. each workstation agent consists of a Network Interface, the Network Interface is described in this section, also because the Generator and the Component Store have a Network Interface as well.

3.4.1 Network Interface (NIN)

Function

For a connected agent, the Network Interface arranges the message reception from and transmission to other agents. It decouples the agents from the Switch Element and vice versa. This way, deadlocks are avoided where the Switch Element and (the Controller of) an agent are waiting for each other.

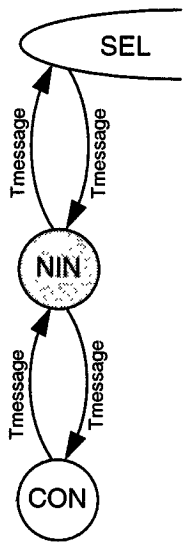


Figure 11 Network Interface

Interface

selnin : various types of messages from the Switch Element
ninsel : various types of messages to the Switch Element
connin : various types of messages from an agent
nincon : various types of messages to an agent

Behaviour

A message received from the Switch Element is stored in a list *selcon*. A message received from the connected workstation agent and to be sent to the Switch Element is given the *id* of the sending agent and is subsequently stored in a list *consel*. As long as both lists are not empty, their contents are sent to the Controller and the Switch Element respectively, and the lists are updated.

Code

```
proc NIN (id : Tid,
              selnin : ? Tmessage, ninsel : ! Tmessage,
              connin : ? Tmessage, nincon : ! Tmessage) =
|[ m : Tmessage,                    -- message
   selcon, consel : Tmessage*       -- lists of messages
```

```
    | selcon := [];consel := [];
   *[ selnin ? m              -> selcon := selcon ++ [m]
    | len (selcon) > 0; nincon ! hd(selcon) -> selcon := tl(selcon)
    | connin ? m       -> m.0 := id; consel := consel ++ [m]
    | len (consel) > 0; ninsel ! hd(consel) -> consel := tl(consel)
    ]
]|
```

3.4.2 Switch Element (SEL)

Function

The Switch Element makes the messages sent by the Network Interfaces arrive at their destinations. The destination is part of the message itself. Every workstation is uniquely identified by a number (see Table II). The destination 0 is a broadcast throughout the network; every connected component receives the broadcasted message, except the Generator, the Component Store, and the sender of the message. Note that the Component Store does not send messages to other agents.

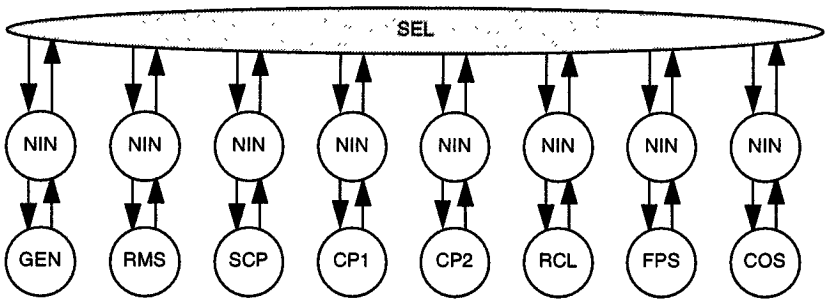


Figure 12 Switch Element

Interface

ninsel(i) : message to the Switch Element from agent *i* (*i* = 1, 2, .., 8; see Table II)
selnin(i) : message from the Switch Element to agent *i* (*i* = 1, 2, .., 8; see Table II)

Behaviour

The variable *address* represents a list of possible addressees. If a message arrives via one of the channels, the Switch Element checks whether it is a broadcast (*m.1* = 0). If it is, *address* is set such that every applicable station except the sender will receive the message. Note that *address.0* is not used. If it is not a broadcast, *address* is set such that only the addressed station will receive the message. Next, for every station is checked whether its corresponding value in *address* is set. If so, the Switch Element sends the incoming message to (the Network Interface of) that agent.

Code

```
proc SEL ( ninsel1, ninsel2, ninsel3, ninsel4, ninsel5, ninsel6, ninsel7, ninsel8:
                                                          ? Tmessage,
           selnin1, selnin2, selnin3, selnin4, selnin5, selnin6, selnin7, selnin8:
                                                          ! Tmessage ) =
|[ m: Tmessage,
   origin, target: nat,                  -- sender, addressee
   address: <bool # bool # bool # bool # bool # bool # bool # bool>
                                         -- list of possible addressees
                                         -- address.0 is not used
 | address := <false, false, false, false, false, false, false, false>;
   *[   [ ninsel1 ? m | ninsel2 ? m | ninsel3 ? m | ninsel4 ? m |
        ninsel5 ? m | ninsel6 ? m | ninsel7 ? m | ninsel8 ? m ];
        origin := m.0;
        [ m.1 = 0 ->
               address := <false, false, true, true, true, true, true, true, false>;
               address.origin := false
        | m.1 /= 0 ->
               target := m.1; address.target := true
        ];
        [ address.1 -> selnin1 ! m | not address.1 -> skip ];      -- GEN
        [ address.2 -> selnin2 ! m | not address.2 -> skip ];      -- RMS
        [ address.3 -> selnin3 ! m | not address.3 -> skip ];      -- SCP
        [ address.4 -> selnin4 ! m | not address.4 -> skip ];      -- CP1
        [ address.5 -> selnin5 ! m | not address.5 -> skip ];      -- CP2
        [ address.6 -> selnin6 ! m | not address.6 -> skip ];      -- RCL
        [ address.7 -> selnin7 ! m | not address.7 -> skip ];      -- FPS
        [ address.8 -> selnin8 ! m | not address.8 -> skip ];      -- COS
```

```
        address := < false, false, false, false, false, false, false, false, false >
    ]
]|
```

3.5 Workstation Agent

The processes that together constitute a workstation, their communication channels, and their environment are represented in Figure 13. The processes and their main tasks are:

- a controller (CON), which distributes messages within the workstation and to other agents;
- a request handler (REQ), which prepares (subcontracting) bids;
- a subcontractor (SUB), which prepares subcontracting task announcements and offers;
- a sender (SEN), which prepares task announcements and task offers;
- a database (DBS), which keeps track of the workstation status;
- a machine controller (MAC), which controls the physical part of the workstation;
- a network interface (NIN), which makes the connection to other agents;
- (the physical part (PHY), which performs the operations).

Note that the network interface is part of the (generic) workstation agent, whereas the physical part (PHY) is not. The characteristics of the latter are obviously dependent on the workstation.

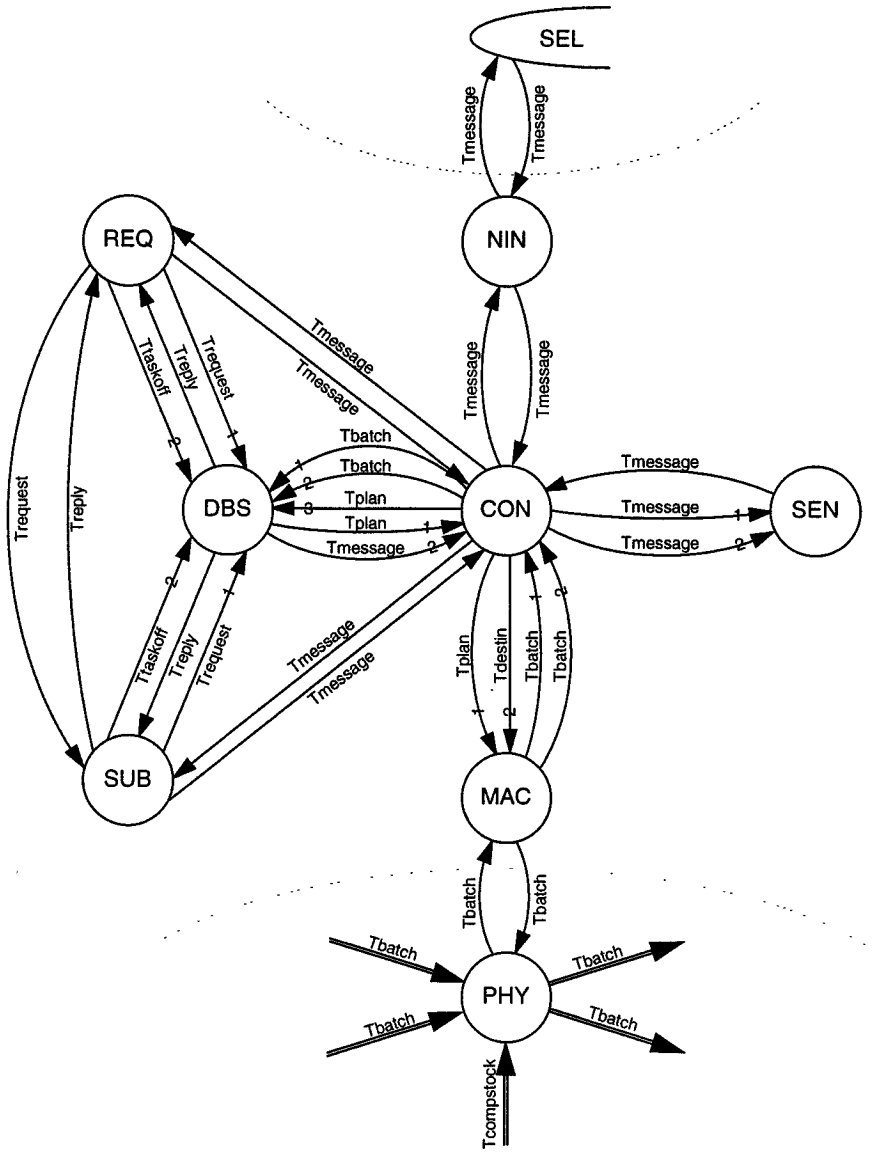


Figure 13 Structure of a workstation

3.5.1 Controller (CON)

Function

The Controller is the coordinator of the various processes. It is also the interface of other processes with the outside world. It stores vital information such as the next destination of the batch in case of subcontracting.

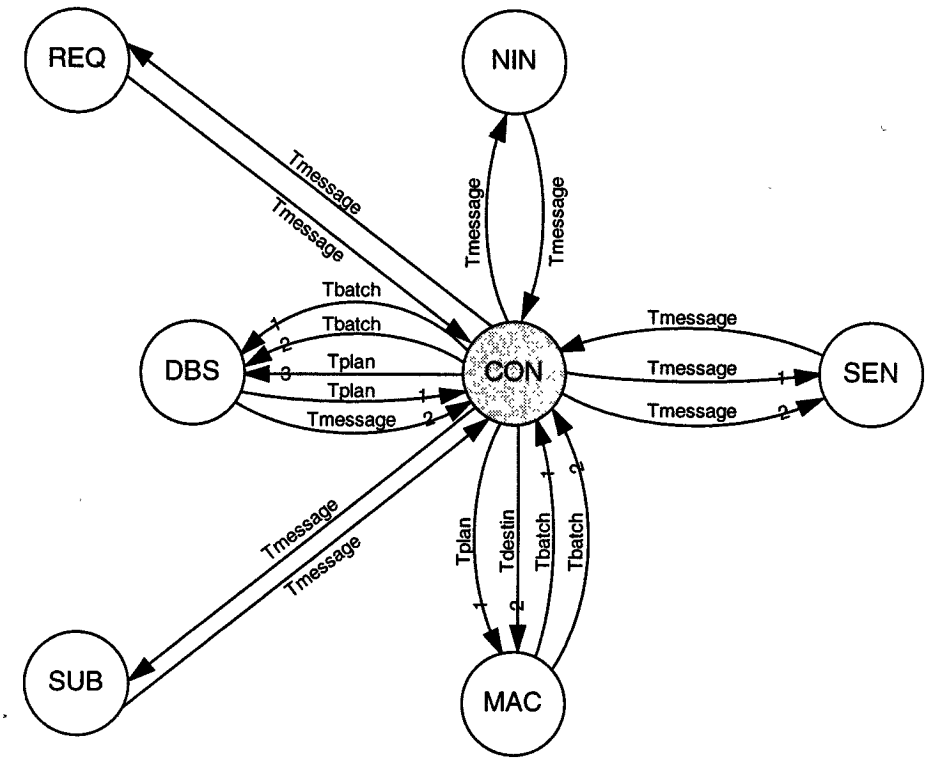


Figure 14 Controller

Interface

connin : various types of messages to other agents
nincon : various types of messages from other agents
conreq : (subcontracting) task announcement, task offer without subcontracting, or subcontracting task offer from another agent
reqcon : (subcontracting) bid to another agent
consub : task offer with subcontracting, or subcontracting bid from another agent
subcon : subcontracting task announcement, or subcontracting task offer to other agents
condbs1 : request for the planning of a newly arrived batch
dbscon1 : planning of a batch
dbscon2 : order to the Component Store to replenish components
condbs2 : notification that operation on the present batch is finished
condbs3 : possible changes in the plan
consen1 : message with information about the next operation
consen2 : bid from other agents
sencon : task announcement, or task offer to other agents
maccon1 : notification of arrived batch
conmac1 : plan for process execution
maccon2 : notification of end of operation
conmac2 : destination for the current batch

Behaviour

(1a) If a message is received from another agent, the Controller forwards this message to another process depending on the type of message:

- *m.2 = 1* or *m.2 = 6*: (subcontracting) task announcements are forwarded to the Request Handler. In addition, the agent checks whether it is sending (*outtaskann*) or should be sending a task announcement itself at the same moment. The agent should be sending a task announcement if a new job should arrive (*plan.1 = time*) and the destination of that job is not determined by subcontracting (*SELDEST(plan.0, xsubdest) = 0*), and the agent is not the last agent in the line (*id /= laststation*).
- *m.2 = 2*: bids without subcontracting are forwarded to the Sender;
- *m.2 = 3*: bids with subcontracting are forwarded to the Sender;
- *m.2 = 4*: task offers without subcontracting are forwarded to the Request Handler;
- *m.2 = 5*: task offers with subcontracting are forwarded to the Subcontractor;
- *m.2 = 7*: subcontracting bids are forwarded to the Subcontractor;
- *m.2 = 8*: subcontracting task offers are forwarded to the Request Handler. Since this agent is offered a subcontracting task offer, it is a subcontractor. Depending on the position of the main contractor, the Controller stores additional information. If the main contractor lies after this agent (*m.0 > id*), this agent as the subcontractor will receive the batch first. When this agent will be finished with its operation, it needs to send the batch to the main contractor, and it does not need to send a task announcement. Therefore, it stores the id (*m.0*) of the main contractor, for application afterwards.
- *m.2 = 9*: job information is not forwarded to other processes. Controllers of the agents forward job information to each other. The job information contains the remaining operations, including the one to be processed at this workstation. Therefore, the Controller filters the next operation and the rest of the operations out of the job information. Later, a message with information about the next operation might be sent to the Sender. Based on the current operation's sequence number, operations are divided into current (*ps.0 = seq*), next (*ps.0 = seq + 1*), and remaining operations (*ps.0 > seq + 1*). Current operations are disregarded, next operations are stored in a list *xnext_op,* and remaining operations are stored in a list *xrest_ps*. Note that the list *xrest_ps* contains all operations except the current and the next one. In case the next destination is known because of subcontracting (*nextdst /= 0*), the list of process steps does not need to be updated. After all, the next agent will work with the same operation as the current one. If the next destination is not known (*nextdst = 0*), the messages with the next and remaining operations are updated. Note that the message with the remaining operations contains all operations except the current one.
- *m.2 = 11*: changes in plans are forwarded to the Database. Based on this message, a plan is created with which the Database might update the agent's schedule. For this, the Database needs to know the first possible finish time, i.e. the first possible start time for the next agent corrected for transportation (*m.8*). The Database responds with a new (possibly updated) schedule for the current job, which is subsequently sent to the Machine Controller. Note that if a schedule update message is received, a batch is always present. The 'next' workstation sends this message if it updates the planning line of a batch, which is not in that workstation yet. It sends the message to the workstation at which the batch is present at that moment.
- *m.2 = 12*: invalid bids without subcontracting are forwarded to the Sender. The Sender will redo the task announcement procedure;
- *m.2 = 13*: invalid bids with subcontracting are forwarded to the Sender. The Sender will redo the task announcement procedure;
- *m.2 = 14*: invalid subcontracting bids are forwarded to the Subcontractor. The Subcontractor will send an invalid bid to the agent that sent the 'original' task announcement.

(2) If a task offer (with or without subcontracting) is received from the Sender (*m.2 = 4 or m.2 = 5*), the destination is set. The fact that a task offer is sent means that there is no outstanding task announcement anymore(*outtaskann := false*) and that the schedule of the agent might need to be updated (*condbs3 ! plan*). The schedule and the task offer are sent to the Machine Controller and to another agent respectively. If a task announcement is received from the Sender, the Controller sends it to other agents via the Network Interface.

(3) If a bid is received from the Request Handler, the Controller checks whether the bid is valid. The bid is valid if it is not made up from invalid subcontracting parts (*m.2 /= 12*), and if the *outtaskann* flag is not set. If the last condition is not fulfilled, the bid is still valid if the agent lies before the agent that sent the task announcement (second-side loop). Finally, the bid is sent to another agent.

(4) If a component order is received from the Database, the Controller forwards it to the Component Store.

(5) If a subcontracting task announcement (*m.2 = 6*) or task offer (*m.2 = 8*) is received from the Subcontractor, the Controller forwards it to (an)other agent(s). If it is a subcontracting task offer, the Controller checks whether it needs to record the next destination. Since this agent offers a subcontracting task offer, it is a main contractor. Depending on the position of the subcontractor, the Controller stores the subcontractor's *id*. If the subcontractor lies after this agent (*m.1 > id*), this agent as the main contractor will receive the batch first. When this agent will be finished with its operation, it needs to send the batch to the subcontractor, and it does not need to send a task announcement. Therefore, it stores the *id* (*m.1*) of the subcontractor, for application afterwards.

(6) If the Machine Controller notifies that a batch is arrived (*maccon1 ? batch*), the Controller forwards the number of the batch to the Database to obtain a schedule for the arrived batch. After the plan is obtained, the destination is checked. If no destination has been determined yet (*destin = 0*) and the workstation is not the last workstation in the factory (*id /= laststation*), the next operation is forwarded to the Sender to make a new task announcement. Otherwise, the plan is sent to the Machine Controller that starts the operation.

(7) If the Machine Controller notifies that it completed the operation (*maccon2 ? batch*), the Controller forwards this information to the Database. The Database will update its schedule by removing the planning line for the current operation. If necessary, the list of subcontracting jobs is updated. The Final Product Store sends a 'finished job' message to the Generator. If the job is not yet finished (*id /= laststation*), the job information, i.e. a message with remaining operations, is sent to the next agent. Finally, the Machine Controller is informed about the next destination. This process will forward the next destination to the Physical System.

Code

```
proc CON (id: nat, laststation: Tid,
          nincon: ? Tmessage, connin: ! Tmessage,
          conreq: ! Tmessage, reqcon: ? Tmessage,
          consub: ! Tmessage, subcon: ? Tmessage,
          consen1, consen2 : ! Tmessage,
          sencon: ? Tmessage,
          conmac1: ! Tplan, conmac2: ! Tdestin,
          maccon1, maccon2: ? Tbatch,
          condbs1: ! Tbatch, condbs2: ! Tbatch, condbs3: ! Tplan,
          dbscon1: ? Tplan,  dbscon2: ? Tmessage, t: real) =
|[ batch: Tbatch,              -- arrival of / start / end operation batch
   plan: Tplan,                -- planning line
   nextdst, destin: Tdestin,   -- next destination
   m,                          -- message
   m_next_op,                  -- message with next operation (process step(s))
   m_rest_ps: Tmessage,        -- message with remaining process steps
   xm_next_op,                 -- list of messages with next operation (process step(s))
   xm_rest_ps: Tmessage*,      -- list of messages with remaining process steps
```

```
   outtaskann: bool,          -- outstanding task announcement?
   seq: Tseqnr,               -- sequence number
   ps: Tprocstep,             -- process step
   xps,                       -- list of process steps
   xnext_op,                  -- next operation (= list of next process step(s))
   xrest_ps: Tprocstep*,      -- list of remaining process steps
   xsubdest: <Tjob # Tdestin>*-- list of subcontract destinations
 | xsubdest := []; xm_next_op := []; xm_rest_ps := []; outtaskann := false;
! time," CON",id," init", nl();
   *[ nincon ? m ->                                                          (1)
            [ m.2 = 1 or m.2 = 6 ->       -- (subcontracting) task announcement
                   [not outtaskann ->
                          condbs1 ! 0;
                          dbscon1 ? plan;
                          outtaskann := (plan.1 = time and id /= laststation and
                                                 SELDEST(plan.0, xsubdest) = 0)
                   | outtaskann -> skip
                   ];
                   conreq ! m;
            | m.2 = 2  -> consen2 ! m            -- bid without subcontracting
            | m.2 = 3  -> consen2 ! m            -- bid with subcontracting
            | m.2 = 4  -> conreq ! m             -- task offer without subcontracting
            | m.2 = 5  -> consub ! m             -- task offer with subcontracting
            | m.2 = 7  -> consub ! m             -- subcontracting bid
            | m.2 = 8  ->                        -- subcontracting task offer
                   [ m.0 > id ->          -- main contractor after this agent
                          xsubdest := xsubdest ++[<m.3, m.0>]
                   | m.0 < id -> skip     -- main contractor before this agent
                   ];
                   conreq ! m
            | m.2 = 9  ->                        -- job information
                   xps := m.7; ps := hd(xps);
                   xnext_op := []; xrest_ps := [];
                   seq := ps.0;
                   *[ len(xps) > 0 ->
                          ps := hd(xps);
                          xps := tl(xps);
                          [ ps.0 = seq -> skip
                          | ps.0 = seq + 1 ->
                                 xnext_op := xnext_op ++ [ps]
                          | ps.0 > seq + 1 ->
                                 xrest_ps := xrest_ps ++ [ps]
                          ]
                   ];
                   xps := xnext_op ++ xrest_ps;
                   m_rest_ps := m; m_next_op := m;
                   nextdst := SELDEST(m.3, xsubdest);
                   [ nextdst = 0 ->                  -- strip receptstappen
                          m_rest_ps.7 := xps;
                          m_next_op.7 := xnext_op;
                   | nextdst /= 0 -> skip            -- subcontracted job
                   ];
                   xm_rest_ps := xm_rest_ps ++ [m_rest_ps];
                   xm_next_op := xm_next_op ++ [m_next_op];
            | m.2 = 11 ->                        -- informs about changes in plans
                   plan := < m.3, 0, m.8, 0, 0, 0, m.7>;
                   condbs3 ! plan;
                   dbscon1 ? plan;
                   conmac1 ! plan
            | m.2 = 12  -> consen2 ! m           -- invalid bid without subcontr.
            | m.2 = 13  -> consen2 ! m           -- invalid bid with subcontracting
            | m.2 = 14  -> consub ! m            -- invalid subcontracting bid
            ]
   | sencon ? m ->                                                           (2)
            [ m.2 = 4 or  m.2 = 5 ->
                   outtaskann := false;
                   destin := m.5;
                   plan := <m.3, 0, m.8 - TRANS(m.5,id), 0, 0, 0, m.7>;
                   m.5 := id;
                   condbs3 ! plan;
                   dbscon1 ? plan;
                   conmac1 ! plan;
            | m.2 = 1 -> skip
            ];
            connin ! m
   | reqcon ? m  ->                                                          (3)
            [ m.2 =  12 -> skip           -- invalid bid (subcontracting part)
            | m.2 /= 12 ->
                   [ outtaskann ->
                          [ m.2 = 2 or m.2 = 3 ->
                                 [ id > m.1 ->
                                        m.2 := m.2 + 10;
                                 | id < m.1 -> skip
                                 ]
                          | m.2 = 7 ->
                                 m.2 := 14;
                          ]
```

```
                    | not outtaskann -> skip
                    ]
            ];
            connin ! m
    | dbscon2 ? m ->                                                    (4)
            connin ! m;
    | subcon ? m ->                                                     (5)
            [ m.2 = 8 ->
                    [ m.1 > id ->          -- subcontractor after this agent
                          xsubdest := xsubdest ++[<m.3, m.1>]
                    | m.1 < id -> skip     -- subcontractor before this agent
                    ]
            | m.2 = 6 -> skip
            ];
            connin ! m
    | maccon1 ? batch  ->                                               (6)
            condbs1 ! batch;
            dbscon1 ? plan;
            destin := SELDEST(batch, xsubdest);
            m_next_op := hd(xm_next_op); xm_next_op := tl(xm_next_op);
            [ destin = 0 and id /= laststation ->
                    m_next_op.8 := plan.2;
                    consen1 ! m_next_op
            | destin /= 0 or id = laststation -> -- FPS, end of operations
                    conmac1 ! plan
            ]
    | maccon2 ? batch ->                                                (7)
            condbs2 ! batch;
            m_rest_ps := hd(xm_rest_ps); xm_rest_ps := tl(xm_rest_ps);
            [ len(xsubdest) > 0 ->
                    [ batch = hd(xsubdest).0 -> xsubdest := tl(xsubdest)
                    | batch /= hd(xsubdest).0 -> skip
                    ]
            | len(xsubdest) = 0 -> skip
            ];
            [ id = laststation ->
                    connin ! <id, 1, 15, batch, 0, 0, 4, [], 0, 0>
            | id /= laststation ->
                    m_rest_ps.0 := id; m_rest_ps.1 := destin; m_rest_ps.2 := 9;
                    m_rest_ps.8 := 0;  m_rest_ps.9 := 0;
                    connin ! m_rest_ps;
                    conmac2 ! destin
            ]
    ]
]|
```

3.5.2 Request Handler (REQ)

Function

The Request Handler handles (subcontracting) task announcements and (subcontracting) task offers. The Request Handler makes bids for task announcements, and it forwards task offers to the Database.

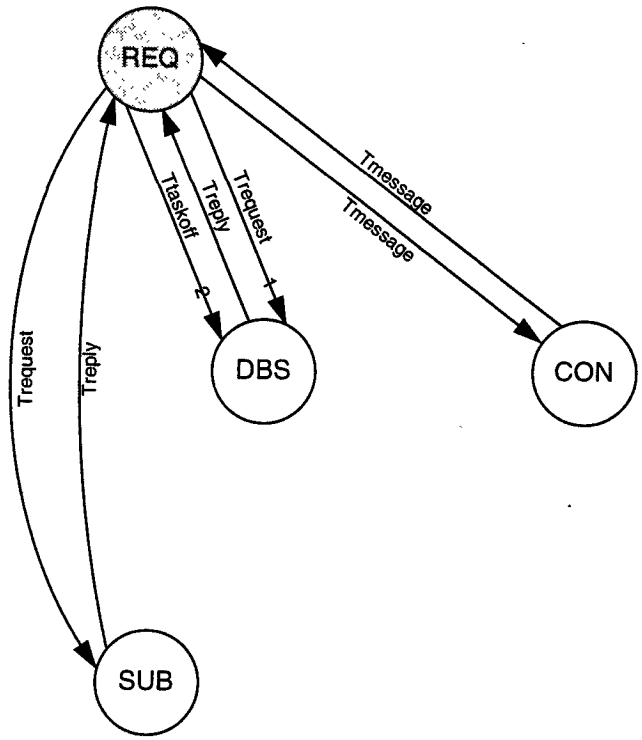


Figure 15 Request Handler

Interface

conreq : (subcontracting) task announcement, task offer without subcontracting, or subcontracting task offer
reqcon : (subcontracting) bid
reqdbs1 : request to calculate the start and end time for a (subcontracting) bid
dbsreq : reply with the start and the end time to prepare a (subcontracting) bid
reqdbs2 : (subcontracting) task offer to be incorporated in the schedule
reqsub : request to subcontract process steps
subreq : reply to prepare a bid with subcontracted process steps

Behaviour

(1) If a task announcement (*m.2 = 1*) is received from another agent, a bid is prepared; a request is sent to the Database to calculate when the operation (*m.7*) would be finished at this agent, given the batch size (*m.6*), data needed to calculate the transport time (*m.5*), and the finish time of the previous workstation (*m.8*) for that job (*m.3*). If the reply from the Database indicates that the agent is not capable of fulfilling the task announcement (*reply.2 = 0*), a bid is not made. Otherwise (*reply.2 /= 0*), the addressee of this agent (*bid.1 := m.0*) and the first possible start and finish time (*bid.8* and *bid.9*) for the bid are set. If the operation contains several process steps (*len(request.3) > 1*), i.e. some process steps might be subcontracted, and if subcontracting is allowed, the same request sent to the Database is forwarded to the

Subcontractor. In case the operation contains only one process step (*len(request.3) = 1*) or subcontracting is not allowed, the remaining information to complete a bid is set: the type (*bid.2 := 2*), and the *id* of the agent that has to receive the batch (*bid.5 := id*). The bid is sent to the Controller by adding it to a list *xm*.

(2) If a task offer without subcontracting (*m.2 = 4*) is received, the task offer is forwarded to the Database that incorporates the new task in the schedule.

(3) The procedure for a subcontracting task announcement (*m.2 = 6*) is almost similar to the procedure for a regular task announcement (see (1)). However, since subcontracting task announcements cannot be subcontracted again, a prepared subcontracting bid is immediately forwarded, i.e. added to list *xm*.

(4) If a subcontracting task offer (*m.2 = 8*) is received, it is forwarded to the Database that incorporates the new task in the schedule.

(5) Process steps of a regular task announcement may be subcontracted (see (1)). If a reply from the Subcontractor is received, the Request Handler checks whether the reply was valid. An invalid reply (*reply.3 = 0*) has two possible causes: either the Subcontracting received invalid subcontracting bids because of outstanding task announcements (*reply.0 = 0*), or the subcontracting and the main tasks did not connect (*reply.0 /= 0*). In the latter case, the agent sends a bid without subcontracting. In the former situation, the agent sends an invalid bid. However, if a valid reply from the Subcontractor is received (*reply.3 /= 0*), the Request Handler makes a comparison between a bid with and a bid without subcontracting. Depending on the outcome of the comparison, the type of the bid is set (*initbid.2 := 3*, a bid with subcontracting; or *initbid.2 := 2*, a bid without subcontracting). In case a bid with subcontracting is better, the planned start and finish time, and the first agent to which the batch should be sent are set (*initbid.5 := reply.3*). Finally, the bid is forwarded to the agent that sent the task announcement by adding the bid to a list *xm*.

(6) The list *xm* decouples the Request Handler from the Controller, just like the Network Interface decouples an agent from the Switch Element. Without this list, deadlocks would occur since the Request Handler and the Controller would both want to send messages to each other at the same time.

Code

```
proc REQ ( id: Tid, subcontracting: bool,
           reqsub: ! Trequest, subreq:  ? Treply,
           reqdbs1: ! Trequest, reqdbs2: ! Ttaskoff, dbsreq: ? Treply,
           conreq:  ? Tmessage, reqcon:  ! Tmessage, t: real ) =
|[ m: Tmessage,
   xm: Tmessage*,                    -- list of messages to CON
   request: Trequest,                -- request to DBS or SUB
   reply: Treply,                    -- reply from DBS or SUB
   bid, subbid,                      -- (subcontracting) bid
   initbid: Tmessage                 -- initial bid/best bid
| xm := [];
*[ conreq ? m ->
              [ m.2 = 1 ->           -- task announcement                    (1)
                     request := <m.3, m.5, m.6, m.7, m.8>;
                     reqdbs1 ! request;
                     dbsreq ? reply;
                     [ reply.2 /= 0 ->
                            bid := m; bid.1 := m.0;
                            bid.8 := reply.1; bid.9 := reply.2;
                            [ len(request.3) > 1 and subcontracting ->
                                   initbid := bid;
                                   reqsub ! request
                            | len(request.3) = 1 or not subcontracting ->
                                   bid.2 := 2; bid.5 := id;
                                   xm := xm ++ [bid]
                            ]
                     | reply.2 = 0 -> skip        -- operation not possible
                     ]
```

```
                | m.2 = 4 ->              -- task offer without subcontracting        (2)
                        reqdbs2 ! m
                | m.2 = 6 ->              -- subcontr. task announcement              (3)
                        request := <m.3, m.5, m.6, m.7, m.8>;
                        reqdbs1 ! request;
                        dbsreq ? reply;
                        [ reply.2 /= 0 ->
                                subbid := m; subbid.1 := m.0; subbid.2 := 7;
                                subbid.8 := reply.1; subbid.9 := reply.2;
                                xm := xm ++ [subbid]
                        | reply.2 = 0 -> skip          -- operation not possible
                        ]
                | m.2 = 8 ->              -- subcontracting task offer               (4)
                        reqdbs2 ! m
                ];
        | subreq ? reply ->                                                         (5)
                [ reply.3 = 0 ->          -- no valid bid from SUB
                        [ reply.0 = 0 ->          -- invalid subcontracting bids
                                initbid.2 := 12;
                        | reply.0 /= 0 ->         -- no valid subcontracting bids
                                initbid.2 := 2; initbid.5 := id;
                        ]
                | reply.3 /= 0 ->
                        [ reply.2 <  initbid.9 ->                -- subcontracting is better
                                initbid.2 := 3; initbid.5 := reply.3
                                initbid.8 := reply.1; initbid.9 := reply.2;
                        | reply.2 >= initbid.9 ->                -- w/o subcontr. is better
                                initbid.2 := 2; initbid.5 := id
                        ]
                ];
                xm := xm ++ [initbid]
        | len(xm) > 0; reqcon ! hd(xm) ->                                           (6)
                        xm := tl(xm)
        ]
]|
```

3.5.3 Subcontractor (SUB)

Function

The Subcontractor issues subcontracting task announcements and awards the best possible subcontractors with subcontracting task offers. Clearly, in the present configuration of the model factory, subcontracting will only take place at the Component Placement stations; only the process step 'component placing' could be needed more than once in an operation, and only for these functions multiple (two) workstations are present. Nevertheless, for reasons of genericity and extensibility, every workstation contains a Subcontractor.

Subcontracting takes place in a rather silly way; the possibility to subcontract every combination of process steps is investigated. A smarter solution might be to offer a subcontract for process steps for which an agent does not have the needed components at its local component store. Other solutions might be to equip agents with knowledge about other agents, but this would violate the prerequisite of agents not having any knowledge about each other.

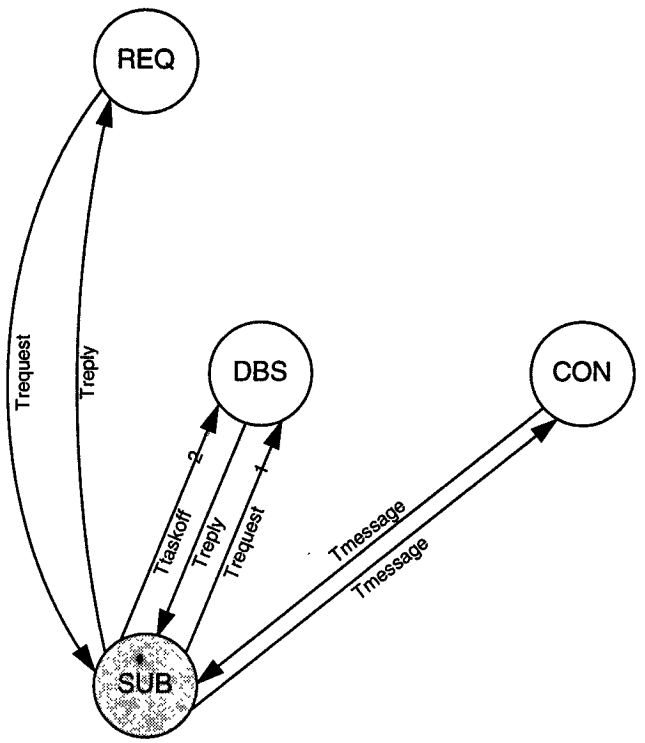


Figure 16 Subcontractor

Interface

consub : task offer with subcontracting, or subcontracting bid
subcon : subcontracting task announcement, or subcontracting task offer
subdbs1 : request to calculate the end time for the main contracting part of a bid
dbssub : reply with the start and end time to prepare the main contracting part of a bid
subdbs2 : main contracting part of a task offer to be incorporated in the planning
reqsub : request to subcontract process steps
subreq : reply to prepare a bid with subcontracted process steps

Behaviour

(1a) If a request to subcontract process steps is received (*reqsub ? request*), initial steps are taken to prepare a bid with subcontracting. After a set of variables is initialised, a number of subcontracting task announcements are prepared in a loop (**[not end -> ....]*). Two lists, *mainxps* and *subxps*, containing the process steps for the main- and subcontractor are initialised and filled. The dummies *j, k,* and *l* indicate the positions *i* of the process steps that are subcontracted in a 'cycle'.

(1b) After this, a request is sent to the Database. In this request, the job only contains the process steps of the main contractor (*mainxps*). When the reply from the Database is received, a task is made and stored in a list. A task consists of the job number, the subjob number, the process steps for the main and subcontractor, and the first possible start and end time. Now, two situations may occur; the position of a subcontractor in the model factory might be in front of or behind the main contractor. Since there is no focused addressing to other agents — task announcements are broadcasted to all other agents —, both possibilities have to be taken into account. Therefore, a main contractor sends two subcontracting task announcements for the same set of subcontracted process steps: one in case the subcontractor is physically in front of the main contractor (even subjob numbers), and the other in case the subcontractor lies behind the main contractor (odd subjob numbers). For the latter case, the first possible start time of the subcontractor has to be determined by calculating the finish time of the main contractor. A request is sent to the Database, and a reply is received. The start time for the subcontracting task announcement is equal to the end time of the main contractor's part plus the transportation time. Then, the subcontracting task announcement is added to a list in order to be broadcasted throughout the network.

(1c) A similar task with increased subjob number is added to the list of tasks. A subcontracting task announcement is defined with a first possible start time equal to the end time of the previous agent (*request.4*). The duration of the main contracting part of the operation is determined later. The task announcement is sent to possible subcontractors by adding it to the list *xm*.

(1d) Finally, some variables are initialised again, and others are reset for the next cycle of the loop. The precise statements are not important here, although they assume that the maximum number of process steps that make up an operation is six. After all, only six components can be placed on a board.

(2a) If a subcontracting bid is received (*m.2 = 7*), the Subcontractor compares this bid with the best bid received so far. Again two situations are possible: a bid is received for a subcontracting task announcement with an odd or even subjob number. For each situation has to be checked whether the sender of the bid is indeed located before or after the main contractor. Situations where agents lying behind the main contractor (*m.0 > id*) send subcontracting bids with even subjob numbers (*m.4 \ 2 = 0*) are ignored. The same holds for situations where agents lying in front of the main contractor (*m.0 < id*) send subcontracting bids with odd subjob numbers (*m.4 \ 2 = 1*).

In case an agent lying in front of the main contractor reacts with a valid subcontracting bid, the main contractor calculates the duration of his part, given the end time of the possible subcontractor. First, the function *SELTASK* retrieves the task concerned from the list of tasks using the job number (*m.3*) and subjob number (*m.4*). The first possible start time of the main contractor is calculated from the end time of the subcontractor (*m.9*). The Database is requested to calculate the execution duration of the main contractor's process steps (*task.2*). Now, the condition is checked whether the main- and subcontracting parts of the operation are connected. In other words, it is not allowed that the first part is executed, and that the batch has

to wait to be transported to the second agent. This condition (*reply.1 - m.9 = TRANS(m.0, id)*) prevents necessary complicated updates later. The answer of the Database is compared with the best subcontracting bid received so far. The latter is updated if the new bid is better. The variable *bestsubcontr* contains the job number, the task stating the distribution of the process steps between main- and subcontractor, the best subcontractor, the start and end time of the main part, and the start and end time of the subcontracting part.

(2b) In case an agent lying behind the main contractor reacts with a valid subcontracting bid (*m.4 \ 2 = 1* and *m.0 > id*), the main contractor does not need to calculate the duration of its part of the operation, since this was done before the subcontracting task announcement was sent. Again, the 'connection condition' is checked, and the received bid is compared with the best subcontracting bid received so far. Again, the latter is updated if the new bid is better.

(3) If a task offer with subcontracting is received (*m.2 = 5*), the Subcontractor requests the Database to plan its own part of the task offer, and it sends a subcontracting task offer to the subcontractor. Both task offers are given the correct values for the process steps (*taskoff.7 := task.2; subtaskoff.7 := task.3*). The addressee and the type for the subcontracting task offer are set. Then, the start and end time for the main and subcontracted parts of the process steps are set. The task offers are forwarded to the Database and the subcontractor respectively.

(4) If an invalid subcontracting bid is received, the variable *invalidbids* is set.

(5) When the time period in which subcontracting bids are received is over, a reply is sent to the Request Handler. If invalid subcontracting bids are received, the Subcontractor sends an empty reply (*<0, 0, 0, 0>*) to the Request Handler. If only valid subcontracting bids are received, but if the 'connection condition' is not fulfilled for any of these bids, the Subcontractor sends an empty reply with the job number. If all conditions are fulfilled, the Subcontractor sends a valid reply consisting of the job number, the first possible (overall) start time, the first possible (overall) finish time, and the agent that should receive the batch first (either the main- or subcontractor). In case this agent will receive the batch first (*id < bestsubcontr.2*), the reply contains the job number, the start time of the main part, the end time of the subcontracted process steps, and the *id* of this agent. In case the subcontractor will receive the batch first, the reply includes the job number, the start time of the subcontracting part, the end time of the main part, and the *id* of the subcontractor.

(6) The list *xm* decouples the Subcontractor from the Controller, just like the Network Interface decouples an agent from the Switch Element. Without this list, deadlocks would occur since the Subcontractor and the Controller would both want to send messages to each other at the same time.

Code

```
proc SUB (id: nat,
          reqsub: ? Trequest, subreq: ! Treply,
          dbssub:  ? Treply, subdbs1: ! Trequest, subdbs2: ! Ttaskoff,
          consub:  ? Tmessage, subcon: ! Tmessage, t: real) =
|[ m: Tmessage,
   taskoff: Tmessage,                  -- task offer to DBS
   subtaskoff: Tmessage,               -- subcontracting task offer to subcontractor
   nr_ps : nat,                        -- number of process steps
   end: bool,                          -- new task, end of possible subcontracts
   nr, i, j, k, l: nat,                -- subjob nr, dummies
   ps: Tprocstep,                      -- process step
   xps, xpsbk,                         -- list of process steps and backup
   mainxps, subxps: Tprocstep*,        -- main and subcontract lists of process steps
   task: Ttask,                        -- task
   xtask: Ttask*,                      -- list of tasks
   xm: Tmessage*,                      -- list of messages to CON
   request: Trequest,                  -- request from REQ
   reply: Treply,                      -- reply from DBS
```

```
  bestsubcontr: <Tjob # Ttask # Tid # Tvalue # Tvalue # Tvalue # Tvalue>,
      -- job, task, station, start time and end time main, start and end time sub)
  timer: real,                          -- timer
  invalidbids,                          -- invalid bids received?
  subtaskann: bool                      -- outstanding subcontracting task announcement?
| subtaskann := false; invalidbids := false; xtask := []; xm := [];
  *[ reqsub ? request ->                                                        (1a)
            bestsubcontr.0 := request.0; bestsubcontr.1 := <0, 0, [], [], 0, 0>;
            bestsubcontr.2 := 0; bestsubcontr.3 := 0; bestsubcontr.4 := 0;
            bestsubcontr.5 := 0; bestsubcontr.6 := 0;
            invalidbids := false; subtaskann := true; timer := time + t;
            xpsbk := request.3; xps := xpsbk;
            nr_ps := len(xps);
            xtask := []; nr := 1;
            i := 1; j := 1; k := 0; l := 0;
            end := false;
            *[ not end ->
                   xps := xpsbk; mainxps := []; subxps := [];
                   *[ i <= nr_ps ->
                          ps := hd(xps); xps := tl(xps);
                          [ (i = j) or (i = k) or (i = l) ->
                                 subxps := subxps ++ [ps];
                          | (i /= j) and (i /= k) and (i /= l) ->
                                 mainxps := mainxps ++ [ps];
                          ];
                          i := i + 1
                   ];
      -- this agent first (odd nrs)                                            (1b)
                   subdbs1 ! <request.0, request.1, request.2, mainxps, request.4>;
                   dbssub ? reply;
                   task := <request.0, nr, mainxps, subxps, reply.1, reply.2>;
                   xtask := xtask ++ [task];
                   m := <id, 0, 6, request.0, nr, id, request.2, subxps, reply.2, 0>;
                   xm := xm ++ [m];
      -- other agent first (even nrs)                                          (1c)
                   nr := nr + 1;
                   task := <request.0, nr, mainxps, subxps, 0, 0>;
                   xtask := xtask ++ [task];
                   m := <id, 0, 6, request.0, nr, request.1, request.2, subxps,
                                                                  request.4, 0>;
                   xm := xm ++ [m];
                   nr := nr + 1;                                                (1d)
                   i := 1; j := j + 1;
                   [ j > nr_ps -> k := k + 1; j := k + 1;
                          [ nr_ps <= 3 -> end := true
                          | nr_ps >  3 -> skip
                          ];
                          [ k > nr_ps - 1 ->
                                 l := l + 1; k := l + 1; j := k + 1;
                                 [ nr_ps <= 5 -> end := true
                                 | nr_ps >  5 -> skip
                                 ];
                                 [ l >  nr_ps - 2 -> end := true
                                 | l <= nr_ps - 2 -> skip
                                 ]
                          | k <= nr_ps - 1 -> skip
                          ]
                   | j <= nr_ps -> skip
                   ]
            ]
| consub ? m ->
      [ m.2 = 7 ->                          -- subcontracting bid               (2)
            [ m.4 \ 2 = 0 ->        -- even nr, so subcontractor first          (2a)
                   [ m.0 < id ->  -- subcontractor located before main contractor
                          task := SELTASK(m.3, m.4, xtask);
                          subdbs1 ! <m.3, m.0, m.6, task.2, m.9>;
                          dbssub ? reply;
                          [ reply.1 - m.9 = TRANS(m.0,id) ->
                                 [ bestsubcontr.4  > reply.2  or bestsubcontr.4  = 0 ->
                                        bestsubcontr.1 := task;
                                        bestsubcontr.2 := m.0;
                                        bestsubcontr.3 := reply.1;
                                        bestsubcontr.4 := reply.2;
                                        bestsubcontr.5 := m.8;
                                        bestsubcontr.6 := m.9
                                 | bestsubcontr.4 <= reply.2 and bestsubcontr.4 /= 0 ->
                                        skip
                                 ]
                          | reply.1 - m.9 /= TRANS(m.0,id) -> skip
                          ]
                   | m.0 > id -> skip     -- this agent located before other agent
                   ]
            | m.4 \ 2 = 1 ->        -- odd nr, so main contractor first         (2b)
                   [ m.0 > id ->  -- main contractor located before subcontractor
                          [ bestsubcontr.6  > m.9  or bestsubcontr.6  = 0 ->
                                 task := SELTASK(m.3, m.4, xtask);
                                 [ m.8 - task.5 = TRANS(m.0,id) ->
```

```
                                        bestsubcontr.1 := task;
                                        bestsubcontr.2 := m.0;
                                        bestsubcontr.3 := task.4;
                                        bestsubcontr.4 := task.5;
                                        bestsubcontr.5 := m.8;
                                        bestsubcontr.6 := m.9
                                    | m.8 - task.5 /= TRANS(m.0,id) -> skip
                                    ]
                                | bestsubcontr.6 <= m.9 and bestsubcontr.6 /= 0 -> skip
                                ]
                            | m.0 < id -> skip    -- other agent located before this agent
                            ]
                    ]
                | m.2 = 5 ->                    -- task offer with subcontracting          (3)
                    taskoff := m; subtaskoff := m; task := bestsubcontr.1;
                    taskoff.7 := task.2; taskoff.8 := bestsubcontr.3;
                    taskoff.9 := bestsubcontr.4;
                    subtaskoff.1 := bestsubcontr.2; subtaskoff.2 := 8;
                    subtaskoff.7 := task.3; subtaskoff.8 := bestsubcontr.5;
                    subtaskoff.9 := bestsubcontr.6;
                    [ subtaskoff.1 < id ->          -- subcontractor first
                        subtaskoff.5 := m.5; taskoff.5 := subtaskoff.1
                    | subtaskoff.1 > id ->          -- main contractor first
                        subtaskoff.5 := id; taskoff.5 := m.5
                    ];
                    subdbs2 ! taskoff;
                    xm := xm ++ [subtaskoff]
                | m.2 = 14 ->                                                          (4)
                    invalidbids := true;
                ]
            | subtaskann; delta timer - time ->                                        (5)
                    subtaskann := false;
                    [invalidbids ->
                        subreq ! <0, 0, 0, 0>;
                        invalidbids := false
                    | not invalidbids ->
                        [ bestsubcontr.6 = 0 -> subreq ! <bestsubcontr.0, 0, 0, 0>
                        | bestsubcontr.6 /= 0 -> skip
                        ];
                        [ id < bestsubcontr.2 ->            -- this agent first
                            subreq ! <bestsubcontr.0, bestsubcontr.3, bestsubcontr.6, id>
                        | id > bestsubcontr.2 ->            -- other agent first
                            subreq ! <bestsubcontr.0, bestsubcontr.5, bestsubcontr.4,
                                                                    bestsubcontr.2>
                        ]
                    ];
            | len(xm) > 0; subcon ! hd(xm) ->                                          (6)
                    xm := tl(xm)
            ]
        ]||
```

3.5.4 Database (DBS)

Function

The Database records information about the planning of operations, and about the available components. With this information, it calculates operation times upon requests by the Request Handler and the Subcontractor. In addition, it updates the planning and the component data when a (subcontracting) task offer has to be incorporated in the schedule. Furthermore, it provides the Controller with plans, for instance for newly arrived batches, and it updates the planning when operation on a batch is finished. Almost all communication with the Database takes place by means of synchronisation, i.e. the client that sends a message to the Database waits for its response before it continues its operation.

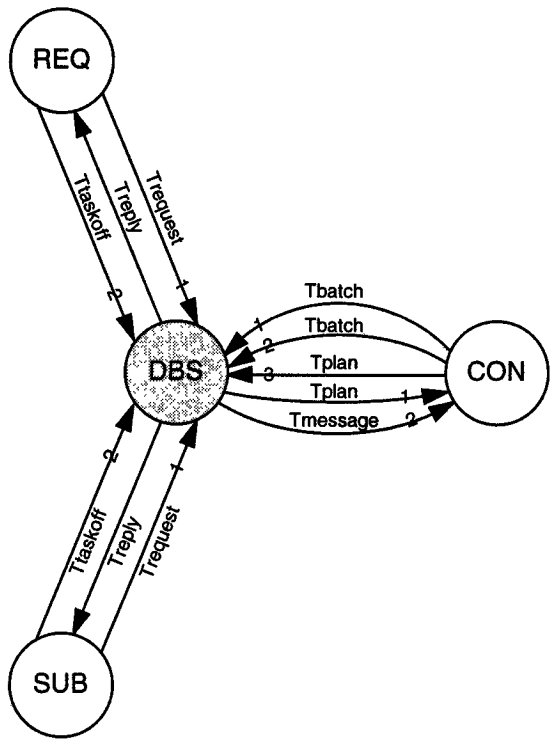


Figure 17 Database

Interface

reqdbs1 : request to calculate the end time for a bid
dbsreq : reply with the end time to prepare a bid
reqdbs2 : task offer to be recorded in the planning
subdbs1 : request to calculate the end time for the main contracting part of a bid
dbssub : reply with the end time to prepare the main contracting part of a bid
subdbs2 : main contracting part of a task offer to be recorded in the planning
condbs1 : request for the planning line of a newly arrived batch
dbscon1 : planning line of a batch
dbscon2 : order to the Component Store to replenish components
condbs2 : notification that operation on the present batch is finished
condbs3 : possible changes in the plan

Behaviour

(1) If a request to calculate the end time for an operation is received from either the Request Handler or the Subcontractor, the channels with these two processes are blocked so that the statements under (2) are carried out before new messages from these processes are received. The origin of the request is stored (1 = Request Handler, 2 = Subcontractor), and a variable indicating a new request is set.

(2) If a new request arrived, the variable *input* stores the *request.* The Database checks by means of the function *OPPOS* whether the agent is capable of performing the requested process steps (*request.3*). In case the requested operation can be executed, the function *PLANNING* calculates the start and the end time for the requested operation. This function is quite complex and takes into account the existing schedule (*xplan*), the characteristics of the new operation (*input*), the available components (*xcompstock*), the operational characteristics of the workstation (*id, xop, fixedoptime*), and the present time (*time*). The fourth element indicates whether it concerns a request or task offer by stating the end time (*0*, i.e. unknown, or *taskoff.9*). The function *PLANNING* returns more than just the start and end time (see (3)). However, here only these times as stored in *plan* are needed. In case the requested operation can not be executed, the reply contains start and end times equal to zero. The reply is sent to the process that sent the request.

(3) If a task offer is received from either the Request Handler or the Subcontractor, the channels with these two processes are blocked so that the statements under (4) are carried out before new messages from these processes are received. A variable indicating a new task offer is set.

(4) When a new task offer was received, the function *PLANNING* is invoked again. This time, all outputs of the function are used except the newly added planning line. After all, this line was used to calculate the end time in (2). The schedule for the workstation and the list of available components are updated (*xplan := z.0,* and *xcompstock := z.3*). To replenish foreseen shortages of components, a replenishment *order* (type 10) is sent to the Component Store. The order is encapsulated in a *Tmessage* format. Each line in the schedule states the previous agent (*plan.4*). All these agents are notified about the new schedule in this agent if the batch is still present at these agents (*time < plan.1 - tin*). The change messages are stored in a list and finally sent to the Controller. Note that actually only the 'previous' agents of *modified* planning lines should be notified.

(5) If a new batch arrives in the workstation, the Controller requests the Database for the planning line for that batch (*batch /= 0*). The Database checks if the first planning line concerns the arrived batch (*batch = plan.0*), and if there is no time error, i.e. whether the batch arrives at the appropriate time. If schedules change, 'previous' agents are notified about the change, except for subcontracted jobs, however. Therefore, a time error might be caused by a subcontracted job of which the schedule was not correctly updated. In that case, the schedule is updated when the batch arrives, and 'previous' agents are notified. Then, the Database sends the first plan to the Controller. Furthermore, the Controller also asks for the current plan (*batch = 0*) to check whether the agent has recently sent a task announcement. In that case, the agent prepares an invalid bid. Therefore, if no planning lines are present, an empty plan is returned.

(6) The Controller notifies the Database, when processing of a batch is finished. Then, the first planning line, i.e. the line for the present batch, is removed from the schedule (*xplan := tl(xplan)*).

(7) If a planning update (*nextplan*) is received from the Controller, the Database checks the plan and determines if it is necessary to update the planning. The update concerns the first

planning line, but for subcontracted jobs, the update might concern the second planning line. Therefore, the first part of the code checks by means of the job *id* (*plan.0*) if the right planning line is updated. If it is necessary to update the schedule (*plan.2 /= nextplan.2*), the function *UPDATEPLANNING* is used. If other planning lines were updated besides the one of the job involved, schedule change messages are sent in order to update the schedules in other agents. Finally the updated planning line is forwarded to the Controller.

(8) The list *xm* decouples the Database from the Controller, just like the Network Interface decouples an agent from the Switch Element. Without this list, deadlocks would occur since the Database and the Controller would both want to send messages to each other at the same time.

Code

```
proc DBS ( id: Tid,
          reqdbs1: ? Trequest, reqdbs2: ? Ttaskoff, dbsreq: ! Treply,
          subdbs1: ? Trequest, subdbs2: ? Ttaskoff, dbssub: ! Treply,
          condbs1: ? Tbatch, condbs2: ? Tbatch, condbs3: ? Tplan,
          dbscon1: ! Tplan, dbscon2: ! Tmessage,
          xop: Topdata*, xcompstock: Tcompstock*, fixedoptime: real) =
|[ chgmes: Tmessage,          -- message for changing the plan in the other agent
   xm: Tmessage*,             -- list of messages
   fplan, plan,               -- first planning line, planning line
   nextplan: Tplan,           -- planning line for next job
   xplan, xplanbk: Tplan*,    -- list of planning lines (backup)
   input,                     -- input data for planning
   request: Trequest,         -- request from REQ,SUB
   reply: Treply,             -- reply to REQ,SUB
   blocked,                   -- channels must be blocked?
   newrequest, newtaskoff,    -- new requests / task offers?
   op_poss: bool,             -- operation possible?
   origin: nat,               -- origin of request (1: REQ, 2: SUB)
   z: <Tplan* # Tplan # Tprocstep* # Tcompstock* # nat>,
       -- new xplan, plan for new request, order, component stock, branch
   y: <Tplan* # Tplan* # nat>,-- new xplan, changed plans, branch
   order: Tprocstep*,         -- component order
   taskoff: Ttaskoff,         -- task offer
   batch: Tbatch,             -- batch
   tin : real                 -- transportation time
 | xplan := []; xplanbk := []; order := []; xm := [];
   blocked := false; newrequest := false; newtaskoff := false;
   *[ not blocked; reqdbs1 ? request ->                                    (1)
            blocked := true; origin := 1; newrequest := true;
    | not blocked; subdbs1 ? request ->
            blocked := true; origin := 2; newrequest := true;
    | newrequest; delta 0 ->                                               (2)
            newrequest := false; input := request;
            op_poss := OPPOS(request.3, xop);
            [ op_poss ->
                  z := PLANNING(xplan, id, input, 0, xcompstock, xop, time,
                                                          fixedoptime);
                  plan := z.1;
                  reply := <request.0, plan.1, plan.2, 0>;
            | not op_poss ->
                  reply := <request.0, 0, 0, 0>;
            ];
            [ origin = 1 -> dbsreq ! reply
            | origin = 2 -> dbssub ! reply
            ];
            blocked := false
    | not blocked; reqdbs2 ? taskoff ->                                    (3)
            blocked := true; newtaskoff := true;
    | not blocked; subdbs2 ? taskoff ->
            blocked := true; newtaskoff := true;
    | newtaskoff; delta 0 ->                                               (4)
            newtaskoff := false;
            input.0 := taskoff.3; input.1 := taskoff.5; input.2 := taskoff.6;
            input.3 := taskoff.7; input.4 := taskoff.8;
            z := PLANNING(xplan, id, input, taskoff.9, xcompstock, xop, time,
                                                          fixedoptime);
            xplan := z.0;
            order := z.2;
            xcompstock := z.3;
            *[ len (z.0) > 0 ->
                  plan := hd(z.0); z.0 := tl(z.0);
                  tin := TRANS(plan.4, id);
                  [ time < plan.1 - tin ->
                        chgmes := < id, plan.4, 11, plan.0, 0, 0, 0, plan.6,
                                                  plan.1-tin, plan.2>;
```

```
                        xm := xm ++ [chgmes]
                    | time >= plan.1 - tin -> skip
                    ]
            ];
            [ len(order) >  0 ->
                    xm := xm ++ [<id, 8, 10, 0, 0, 0, 4, order, 0, 0>]
            | len(order) = 0 -> skip
            ];
            blocked := false
    | condbs1 ? batch ->            -- arrival physical batch                    (5)
      [ len(xplan) > 0 ->
            plan := hd(xplan);
            [ batch /= 0 ->
                    [ batch /= plan.0 ->
                        !time, "DBS",id," error in planning", nl();
                    | batch =  plan.0 ->
                        [ time /= plan.1 ->
                            !time, "DBS",id," warning, time error in planning,
                                                    subcontracting??", nl();
                            y := UPDATEPLANNING3(xplan, time, id);
                            *[ len (y.1) > 0 ->
                                plan := hd(y.1); y.1 := tl(y.1);
                                tin := TRANS(plan.4, id);
                                chgmes := < id, plan.4, 11, plan.0, 0, 0, 0,
                                                    plan.6, plan.1-tin, plan.2>;
                                xm := xm ++ [chgmes];
                            ];
                            xplan := y.0; plan := hd(xplan);
                        | time  = plan.1 -> skip
                        ]
                    ]
            | batch = 0 -> skip
            ];
      | len(xplan) = 0 ->
            plan := <0, 0, 0, 0, 0, 0, []>
      ];
      dbscon1 ! plan
    | condbs2 ? batch ->            -- operation on batch is finished            (6)
            xplan := tl(xplan)
    | condbs3 ? nextplan ->         -- planning update                           (7)
            plan := hd(xplan); xplan := tl(xplan); fplan := plan; fplan.0 := 0;
            [ nextplan.0 /= plan.0 ->
                    [ len(xplan) > 0 ->
                        fplan := plan; plan := hd(xplan); xplan := tl(xplan);
                        [ nextplan.0 /= plan.0 ->
                            !time, "DBS",id," error in planning 3", nl()
                        | nextplan.0 = plan.0 ->
                            !time, "DBS",id," warning, subcontracted
                                                                job??",nl()
                        ]
                    | len(xplan) = 0 ->
                        !time, "DBS",id," error in planning 3", nl()
                    ]
            | nextplan.0 = plan.0 ->
                    skip
            ];
            [ plan.2 = nextplan.2 ->
                    xplan := [plan] ++ xplan
            | plan.2 /= nextplan.2 ->
                    plan.2 := nextplan.2;
                    xplan := [plan] ++ xplan;
                    [ len(xplan) > 1 ->
                        y := UPDATEPLANNING(xplan,id);
                        xplan := y.0;
                        *[ len (y.1) > 0 ->
                            plan := hd(y.1); y.1 := tl(y.1);
                            tin := TRANS(plan.4, id);
                            chgmes := < id, plan.4, 11, plan.0, 0, 0, 0,
                                                    plan.6, plan.1-tin, plan.2>;
                            xm := xm ++ [chgmes];
                        ];
                    | len(xplan) = 1 -> skip
                    ];
            ];
            [ fplan.0 =  0 -> skip
            | fplan.0 /= 0 -> xplan := [fplan] ++ xplan
            ];
            plan := hd(xplan);
            dbscon1 ! plan
    | len(xm) > 0; dbscon2 ! hd(xm) ->                                           (8)
            xm := tl(xm)
    ]
]|
```

3.5.5 Sender (SEN)

Function

If the job that is about to be executed is not subcontracted, the agent has to find another agent that will execute the next operation. Therefore, the Sender issues task announcements, evaluates incoming bids, and offers a task to the agent that sent the best bid.

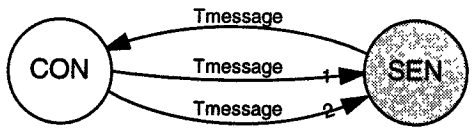


Figure 18 Sender

Interface

consen1 : message with information needed to prepare the next task announcement
consen2 : bid from another agent
sencon : task announcement, or task offer to (an)other agent(s)

Behaviour

(1) The Controller sends a message *taskann* with information about the operation to be executed by a subsequent agent. The Sender sets the variables to make a new task announcement, and it defines the correct task announcement (*taskann.2 = 1*) to be broadcasted (*taskann.1 = 0*) to other agents via the Controller.

(2) Agents that are capable of executing the next operation as defined in the Sender's task announcement reply with bids *m*. The Sender receives bids from these other agents via the Controller and channel *consen2*. In case the incoming bid is a valid bid (*m.2 = 2 or m.2 = 3*), it replaces the best bid received so far if the projected end time for execution of the next operation lies before that of the best bid (*bestbid.9 > m.9*) or if the best bid is the initial bid (*bestbid.9 = 0*). In case the incoming bid is an invalid bid (*m.2 = 12 or m.2 = 13*), the variable *invalidbids* is set; there is no need to compare the incoming bid with the best bid.

(3) When the negotiation time for the outstanding task announcement is over (*delta (timer - time)*), the best bid has been determined. If, however, invalid bids were received (*invalidbids = true*), the whole task announcement procedure has to be redone. If only valid bids were received, the Sender selects the best bid. Recall that always at least one bid will be received. If the best bid is a bid without subcontracting (*bestbid.2 = 2*), a task offer without subcontracting (*taskoff.2 := 4*) is sent. If the best bid involves subcontracting (*bestbid.2 = 3*), a task offer with subcontracting (*taskoff.2 := 5*) is sent. The task offer is forwarded (*sencon ! taskoff*) to the agent that issued the bid. This is the sign for the Controller to start execution on the batch.

(4) When the time has come to send a new task announcement or to re-send the previous task announcement, the best bid and some other variables are defined. Then, the Sender sends the task announcement to the Controller. Note that there is no list *xm* needed to decouple the Sender and the Controller; these two processes do not send messages to each other at the same moment in time.

Code

```
proc SEN ( id: Tid,
           consen1, consen2: ? Tmessage, sencon: ! Tmessage,
           t: real ) =
```

```
|[ m,                                  -- message
   taskoff,                            -- task offer
   taskann,                            -- task announcement
   bestbid: Tmessage,                  -- best bid
   sendnewtaskann,                     -- new task announcement needed?
   invalidbids,                        -- invalid bids received?
   outtaskann: bool,                   -- outstanding task announcement?
   sendnewtasktimer,                   -- timer for new task announcement
   timer: real                         -- timer for outstanding task announcement
 | outtaskann := false; sendnewtaskann := false;
   *[ consen1 ? taskann ->                                                    (1)
              sendnewtaskann := true;
              sendnewtasktimer := time;
              taskann.1 := 0; taskann.2 := 1;
              taskann.5 := id; taskann.9 := 0;
    | consen2 ? m ->                                                          (2)
              [ m.2 = 2 or m.2 = 3 ->
                     [ bestbid.9 >  m.9 or  bestbid.9 =  0 -> bestbid := m
                     | bestbid.9 <= m.9 and bestbid.9 /= 0 -> skip
                     ];
              | m.2 = 12 or m.2 = 13 ->
                     invalidbids := true;
              ];
    | outtaskann; delta (timer - time) ->                                     (3)
              outtaskann := false;
              [invalidbids ->
                     sendnewtaskann := true;
                     sendnewtasktimer := time + t;
              | not invalidbids ->
                     taskoff := bestbid; taskoff.1 := bestbid.0;
                     taskoff.2 := bestbid.2 + 2;
                     sencon ! taskoff
              ]
    | sendnewtaskann; delta (sendnewtasktimer - time) ->                      (4)
              sendnewtaskann := false;
              invalidbids := false; outtaskann := true;
              bestbid := <0, 0, 0, taskann.3, 0, 0, 0, [], 0, 0>;
              timer := time + 2 * t;
              sencon ! taskann
   ]
]|
```

3.5.6 Machine Controller (MAC)

Function

The Machine Controller controls the physical manufacturing system. It transports notifications of batch arrivals detected by sensors of the Physical System to the Controller. It starts execution of the operation as commanded by the Controller, and makes the Physical System send the batch to the determined next destination.

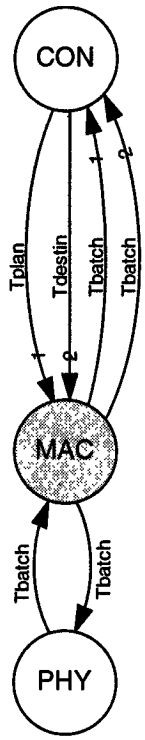


Figure 19 Machine Controller

Interface

phymac : notification of arrived batch at the Physical System
maccon1 : notification of arrived batch
conmac1 : plan for process execution
maccon2 : notification of end of operation
conmac2 : destination for the current batch
macphy : destination for the current batch

Behaviour

(1) If a notification of a batch arrival is received from the Physical System, the Machine Controller informs the Controller about it.

(2) If a planning line is received from the Controller, the workstation starts processing (if it was not doing so already), and the end time is (re-)set.

(3) If process execution is finished, i.e. *delta (endtime - time)* becomes zero, the status of the Machine Controller is reset, and the Controller is informed that the operation on the batch is finished.

(4) If the next destination for the current batch is received from the Controller, it is sent to the Physical System that actually forwards the batch to a conveyor.

Code

```
proc MAC ( id: nat,
           conmac1: ? Tplan, conmac2: ? Tdestin,
           maccon1: ! Tbatch, maccon2: ! Tbatch,
           phymac: ? Tbatch, macphy: ! Tdestin ) =
|[ batch: Tbatch,                        -- batch
   endtime: real,                        -- finish time
   destin: Tdestin,                      -- destination
   plan: Tplan,                          -- plan
   inprocess: bool                       -- in process
 | inprocess := false;
   *[ phymac ? batch  ->                                                (1)
              maccon1 ! batch
    | conmac1 ? plan   ->                                               (2)
              inprocess := true; endtime := plan.2;
    | inprocess; delta (endtime - time) ->                              (3)
              inprocess := false;
              maccon2 ! batch
    | conmac2 ? destin ->                                               (4)
              macphy ! destin
    ]
]|
```

3.6 Physical Part of a workstation (PHY)

Function

The Physical System performs the actual operations on batches. It receives batches, performs the process steps, and sends the batches to other workstations. It is the only (structural) part of a workstation that is specific for that workstation; all other processes are generic. The specific features of a workstation are characterised by the conveyor belts that connect individual workstations (see section 3.8).

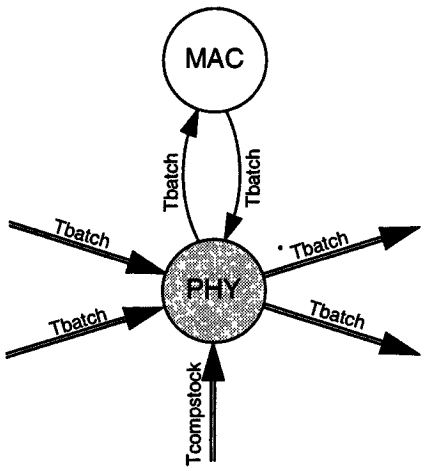


Figure 20 Physical System

Interface

phymac : notification of arrived batch
macphy : destination for the current batch
trxyyy : incoming channel number *x* at station *yyy* (see section 3.8)
trxyyy : outgoing channel number *x* at station *yyy* (see section 3.8)
cosphy : supply of components from the Component Store

Behaviour

(1) The Physical part of the Component Placer 2 is taken as an example. It receives batches from two incoming channels: *tr2cp2*, which is connected to the Conveyor between the Screen Printer and Component Placer 2, and *tr5cp2*, which is connected to the Conveyor between the Component Placer 1 and Component Placer 2. If a batch via either channel arrives, the Physical System informs the Machine Controller about it.

(2) If a destination is received from the Machine Controller, the batch is sent via an outgoing channel to the Conveyor. In this case, batches are always sent via channel *tr6cp2*, the conveyor that connects the Component Placer 2 and the Reflow and Cleaning station.

(3) If components are received from the Component Store, nothing happens with them. In reality, these components would be stored and used during operation. However, in this model the Database records changes in the (economic) stock level. For the moment, nothing is being done with the arrived components.

Code

```
proc PHYCP2 ( tr2cp2: ? Tbatch, tr5cp2: ? Tbatch, tr6cp2: ! Tbatch,
              macphy: ? Tdestin, phymac: ! Tbatch,
```

```
            cosphy: ? Tcompstock) =
|[ batch: Tbatch,
   destin: Tdestin,
   compstock: Tcompstock
 | *[ tr2cp2 ? batch -> phymac ! batch                                    (1)
    | tr5cp2 ? batch -> phymac ! batch
    | macphy ? destin ->                                                  (2)
              [ destin =  6 -> tr6cp2 ! batch
              | destin /= 6 -> !time," error in PHYCP2"
              ]
    | cosphy ? compstock ->                                               (3)
              skip
    ]
]|
```

3.7 Component Store (COS)

Function

The Component Store supplies workstations with components upon request. It always sends trays of four components each. The type of all four components is the same; they all are either yellow, red, or green. The Component Store is only physically connected to workstations that actually need components to perform their operations (see section 3.9). Note that the channels from the Component Store to the Network Interface, and from the Network Interface to the Switch Element are not used; the Component Store does not send messages to other workstations.

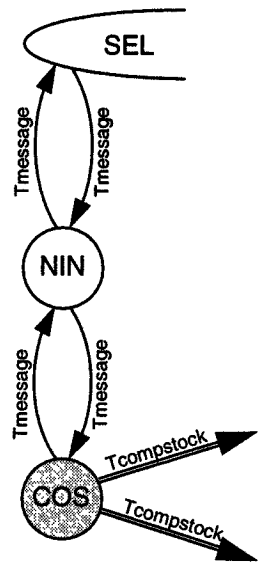


Figure 21 Component Store

Interface

nincos : incoming order to replenish components
cosnin : not applicable
tr9cos : outgoing channel to Component Conveyor of Component Placer 1 (see section 3.9)
tr10cos : outgoing channel to Component Conveyor of Component Placer 2 (see section 3.9)

Behaviour

(1) If a replenishment order is received via the Switch Element and the Network Interface, the Component Store determines the time it takes to conclude the order (*ct*), and adds this message that encapsulates the order to a list *delivery*. However, workstations do not actually physically need components to perform their operations; they only register the number of components in the Database, whether they are present or not.

(2) When the components for the first order are ready, the components are actually delivered. The order is stored in a message *ordermessage* in such way that it may contain multiple lines. Each order line which consists of a number of components (*ordermessage.5*) of a specific type (*orderline.1*) is individually sent to the 'customer' (*ordermessage.0*).

Code

```
proc COS (id: Tid, selcos: ? Tmessage, tr9cos, tr10cos: ! Tcompstock, ct : real ) =
|[ ordermessage,                    -- dummy
```

```
  m: Tmessage;                          -- message
  orderline: Tprocstep,                 -- component order line
  order: Tprocstep*,                    -- component order
  delivery: <Tmessage # real>*          -- component delivery
| order := []; delivery := [];
  *[ selcos ? m ->                                                              (1)
           delivery := delivery ++ [<m, time + ct>];
   | len(delivery) > 0; delta hd(delivery).1 - time ->                          (2)
           ordermessage := hd(delivery).0; delivery := tl(delivery);
           order := ordermessage.7;
           *[ len(order) > 0 ->
                  orderline := hd(order);
                  order := tl(order);
                  [ ordermessage.0 = 4 ->
                        tr9cos ! <orderline.1, ordermessage.5>
                  | ordermessage.0 = 5 ->
                        tr10cos ! <orderline.1, ordermessage.5>
                  ]
           ]
  ]
]|
```

3.8 Conveyor (CVY) and Component Conveyor (CCO)

Function

Conveyors transport batches between workstations. Each conveyor connects two workstations. The transportation duration is determined by the distance between the two workstations, i.e. it is proportional to the absolute difference between the two workstation identification numbers. The two Component Conveyors connect the Component Store with the two Component Placement workstations (see section 3.9).

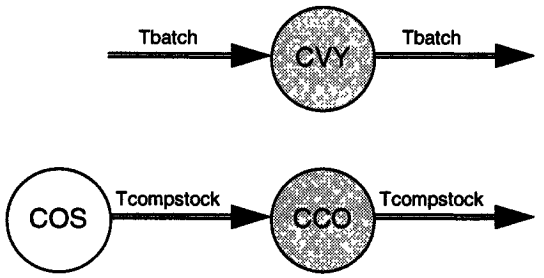


Figure 22 Conveyors

Interface

in : incoming batch/components
out : outgoing batch/components

Behaviour

(1) If a batch or component tray is received, the Conveyor waits a certain time period. Then it sends the batch or component tray to the next station. However, since the component conveyor might have to transport multiple component trays at the same time, a slightly different procedure is needed.

Code

```
proc CVY ( in : ? Tbatch, out : ! Tbatch, ct : real) =
|[ batch : Tbatch
 | *[  in ? batch -> delta ct; out ! batch ]                                    (1)
]|

proc CCO ( in : ? Tcompstock, out : ! Tcompstock, ct : real) =
|[ compstock : Tcompstock, xcomp : <Tcompstock # real>*
 | *[ in ? compstock -> xcomp := xcomp ++ [<compstock, time + ct>]               (1)
    | len(xcomp) > 0; delta (hd(xcomp).1 - time) ->
              out ! hd(xcomp).0; xcomp := tl(xcomp)
    ]
]|
```

3.9 Overall Physical System

The way the individual workstations are connected determines the 'configuration' of the model factory. It also greatly determines the suitability of an agent based control system. Workstations are connected by means of conveyor belts that transport boards between workstations. In total, nine conveyor belts for boards and two for component trays are present. Figure 23 shows the physical system of the model factory. Each arrow represents a conveyor.

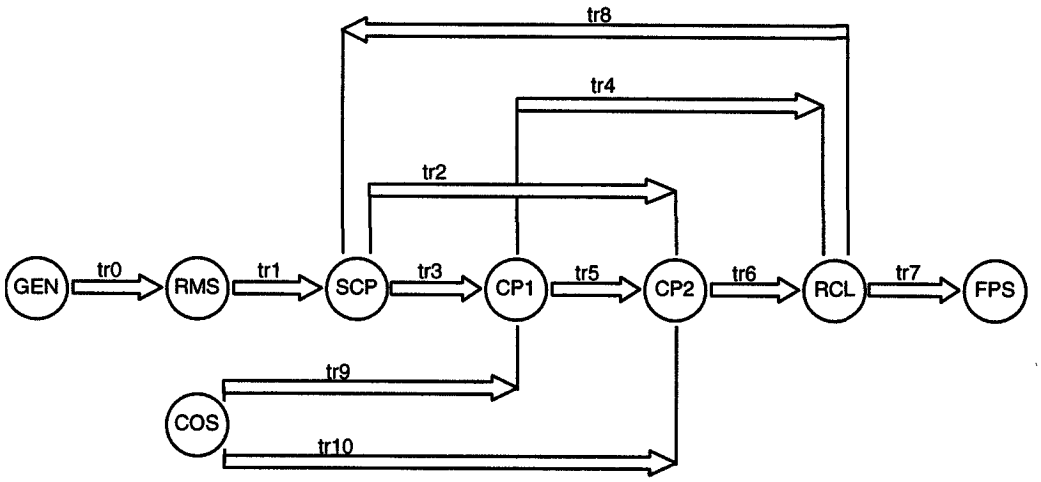


Figure 23 Overall Physical System

4. Simulation results Agent Based Control model

4.1 Introduction

Before the simulations are described, it is necessary to explain the following definitions:

- The 'throughput time' is the time needed to complete a sample of jobs.
- The 'system capacity' is the maximum number of jobs that are simultaneously being processed by the workstations in the model factory. If the maximum capacity is reached, the Generator will not dispatch the next job before a job in the system is finished. This way, a kind of input/output control is realised.
- In all experiments, twenty samples are simulated. In each sample, the number of jobs being processed is 100.

This report focuses on the most important experiments. Obviously, many more interesting experiments could easily be done with small modifications of the models. Especially, readers interested in the logistical aspects could find easy ways to improve the models.

4.2 Influence of system capacity

In the first experiment, the influence of the system capacity on the throughput time is determined. The system capacity is varied, and the throughput times are measured. The specific characteristics of this experiment are as follows:

- The model factory contains the workstations: Raw Material Store, Screen Printing, Component Placement 1, Component Placement 2, Reflow and Cleaning, and Final Product Store. In addition, a Component Store and a Generator are present.
- The system capacity is variable.
- The transportation time for batches between workstations is equal to 5 times the absolute difference between the workstations concerned.
- The component stocks at the component placers are initially completely filled with four components of each type.
- The probability of a second side loop is 20%.
- Twenty samples of 100 jobs are simulated.
- Table VI displays the appropriate operation and replenishment times.

Table VI Standard operation times and component delivery duration

Work-station	Type of operation	Processing time per batch	Variable operation time per board/component	Component delivery duration
RMS	board dispatching	0	10	-
SCP	screen printing	0	20	-
CP1	placing components	0	20	5
CP2	placing components	0	20	5
RCL	reflow & cleaning	0	20	-
FPS	final product store	0	60	-

The results of the experiment are shown in Table VII and Figure 24. Obviously, the average throughput time decreases if the maximum number of jobs in the system increases. However, there seems to be a minimum at a system capacity of 5. It is not clear whether this is a purely random minimum or not. The experiment was probably not large enough to determine whether Figure 24 should exhibit a non-increasing line, or that indeed five jobs is the optimal number of jobs in the system.

Table VII System capacity and throughput time

System capacity	Average throughput time
1	41623.5
2	22600.5
3	16965.8
4	14538.5
5	13831.5
6	13856.8

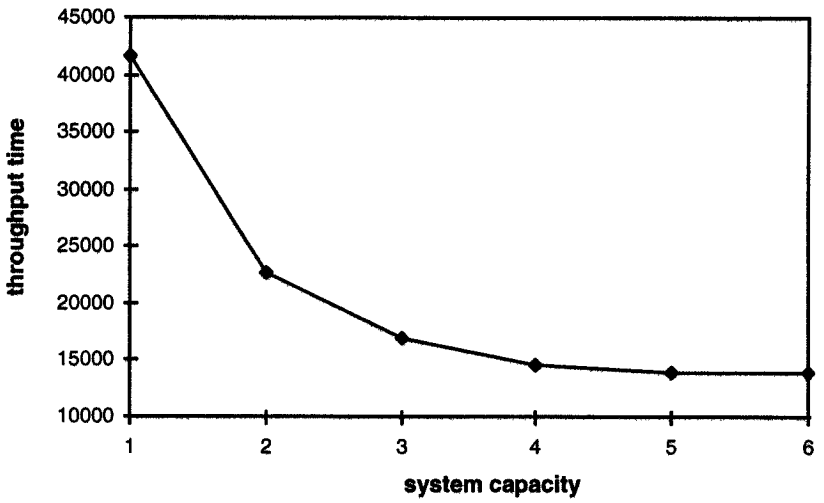


Figure 24 System capacity and throughput time

The following figure shows the capacity utilisation of the six workstation in the model factory. Note that in Figure 25 the lines for the Screen Printer and the Reflow & Cleaning station coincide, since both stations have the same operational characteristics. Only the lines for the component placers have real meaning. The other lines are proportional to each other, since the operation times of these stations per board are fixed. The operation 'component placement' takes most time. However, two component placement stations are present, so that jobs are distributed among them. Therefore, on average the bottleneck is the last station in the line, the Final Product Store. The lines for the two component placement stations come together. With low system capacity, the bids from CP1 will be better than those from CP2, simply because of the transportation time. Note that this is a form of suboptimisation. This situation is comparable to a roadway consisting of two lanes; if traffic is low, everybody will take the right lane, and if there is a lot of traffic both lanes will be equally used.

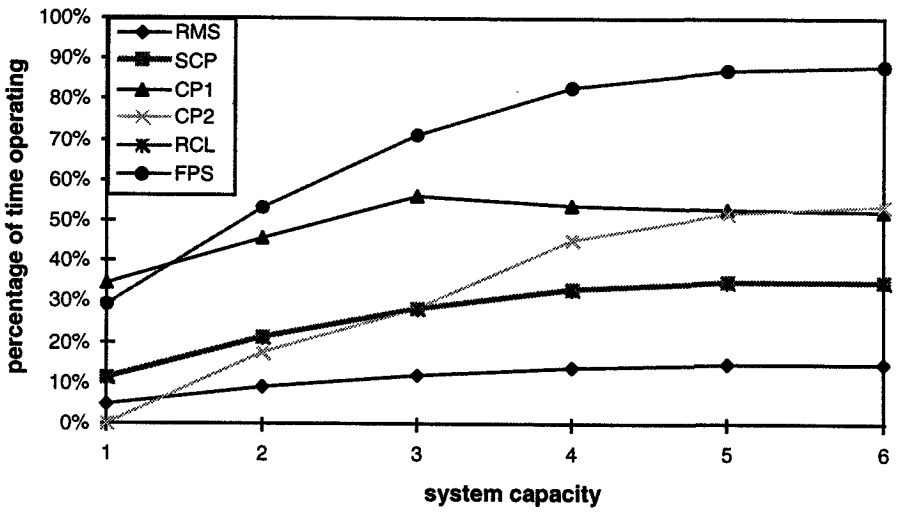


Figure 25 System capacity and capacity utilisation

With increasing system capacity, the risk for deadlocks increases as well. No deadlock occurred in the samples with a system capacity of four or smaller. However, three and nine deadlocks occurred with a system capacity of five and six respectively.

Let us have a closer look at the deadlocks. It appears that all deadlocks result from the same reason: a job A has to wait until it can go to the next station, but since A has to wait B has to wait, and therefore C has to wait, and finally because of the second-side loop A has to wait, and so on. Figure 26 shows such a situation. At the moment job 19 arrives in CP2, this station sends a broadcast to all other stations. RCL replies with a bid, the bid is awarded, and job 19 is planned at RCL after job 18. This means that job 19 has to wait in CP2 until job 18 is finished in RCL, and job 19 can go to RCL. Since job 19 has to wait in CP2, job 20 is postponed, and has to wait in SCP. Since job 20 has to wait in SCP, job 17 is postponed, and has to wait in RCL. Finally, since job 17 has to wait, jobs 18 and the newly planned job 19 are postponed. The result is a deadlock. Obviously, the deadlock could be avoided, for instance by re-allocating job 20 to CP1. A scheduler which would have an overview of the complete process could easily do this.

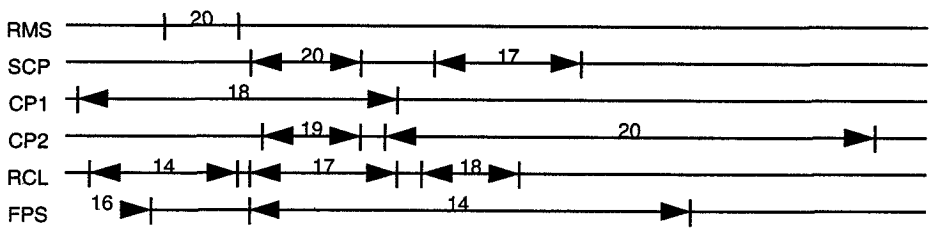


Figure 26 Deadlock

In the experiments in this section, there is no difference between a model where subcontracting is allowed, and a model where it is not. The reason is that the time needed to replenish the component stock is always (much) lower than the time needed to process a batch in the Screen Printing station and to transport it to the next workstation. Consequently, bids with subcontracting will always be worse than 'normal' bids.

4.3 Influence of subcontracting

In this experiment, the effect of subcontracting is shown. The following changes are made compared to the previous experiment:

- The system capacity is four.
- The component replenishment time is variable.

The results of the experiment with the same samples as in the previous experiment are shown in Table VIII and Figure 27. There is not much difference in the results of the experiments with and without subcontracting. Subcontracting is only better if it prevents the replenishment of components, and if replenishment is needed in a situation without subcontracting. However, the replenishment time that is 'saved' for a Component Placement station, has to be 'used' for one of the next jobs that is allocated to the same station. Subcontracting or not, the total number of components and therefore the total number of replenishment times for a sample is roughly equal for both situations. For a particular sample, the experiment with subcontracting can only 'save' two replenishment times at most, namely one at each component placement station.

The results could be improved by more intelligent algorithms. For example, a certain safety stock could be used, and components could be ordered if their number drops below a certain level.

Table VIII Subcontracting, replenishment time and throughput time

Replenishment time	With subcontracting	Without subcontracting
5	14538.5	14538.5
25	14538.5	14538.5
50	14584.8	14624.3
75	14854.1	14980.8
100	15423.4	15582.5
150	17498.1	17773.4
200	20394.0	20958.9
250	24980.9	25411.1
500	48368.0	48808.8

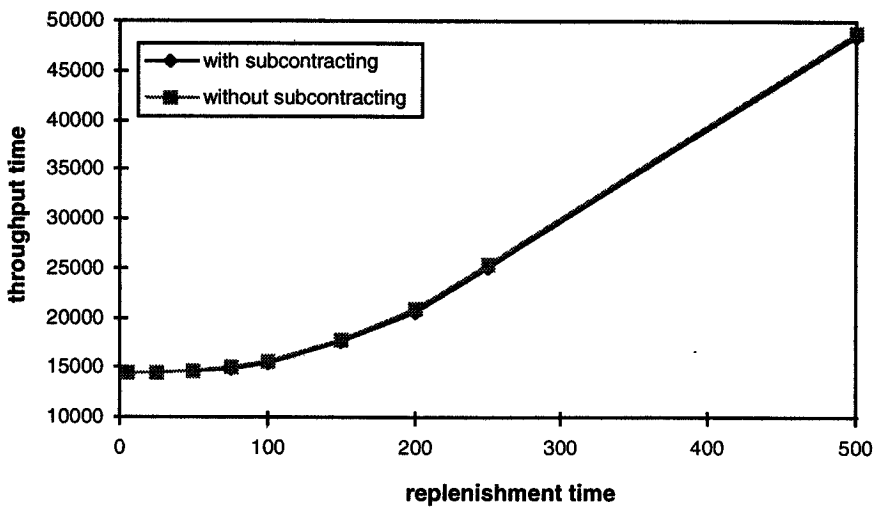


Figure 27 Subcontracting situation

The effect of the possibility to subcontract jobs is bipartite: the number of message exchanges in the system increases tremendously, and the performance is marginally improved. The overall effect of subcontracting is rather negative.

4.4 Influence of negotiation

In this experiment, the effect of negotiation is shown. The following changes are made compared to the first experiment:

- There is no negotiation between workstation agents.

Since there is no more negotiation between workstations, it has to be decided which workstation is going to execute which operation. More precisely, the component placement operations are allocated to the component placers by the Generator. For this, the third attribute in type Tprocstep = < Tseqnr # Top # Tinfo > is used.

Other changes that have to be made in the code are:

- Controller: if a job information message is received, the Controller has to extract the process steps for the next operation, and all remaining operations.
- Sender: if information about the next operation comes in, the Sender constructs a task announcement and sends it to the agent to which the operation was allocated, instead of to all agents.

The results of the experiment are shown in Table IX and Figure 28. The figure also presents the results with subcontracting (similar to Figure 24), and shows a clear difference in the results with and without negotiation. Again, the average throughput time decreases if the maximum number of jobs in the system increases. And again, there seems to be a minimum at a system capacity of five, although this result is hardly valid due to the large number of deadlocks. With a system capacity of three or less, no deadlocks occur. Four deadlocks occur with a system capacity of four. Only four (out of twenty) samples do not lead to a deadlock, if the system capacity is five or six.

Table IX Negotiation, system capacity and throughput time

System capacity	With negotiation	Without negotiation
1	41623.5	41623.5
2	22600.5	23746.3
3	16965.8	18071.5
4	14538.5	15796.9
5	13831.5	15105.0
6	13856.8	15115.0

Again, let us have a closer look at the deadlocks. Again, it appears that most deadlocks result from the same reasons: a job A has to wait until it can go to the next station, but since A has to wait B has to wait, and therefore C has to wait, and finally A has to wait because of the second-side loop, and so on. Figure 29 shows such a situation. It is the first sample which shows a deadlock with a system capacity of four. At the moment job 22 arrives in CP2, this station sends a task announcement to RCL. RCL replies with a bid, the bid is awarded, and job 22 is planned after job 21. This means that job 22 has to wait in CP2 until job 21 is finished in RCL, and job 22 can go to RCL. Since job 22 has to wait in CP2, job 23 is postponed, and has to wait

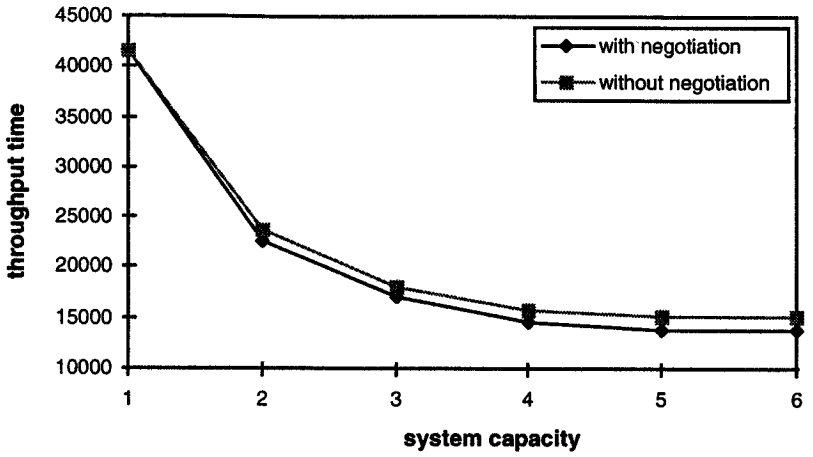


Figure 28 Negotiation, system capacity, and throughput time

in SCP. Since job 23 has to wait in SCP, jobs 24 and 21 are postponed. Thus, job 21 has to wait in RCL. Finally, since job 21 has to wait, the newly planned job 22 is postponed. The result is a deadlock. Obviously, the deadlock would not occur if negotiation was applied. Then, job 23 would be allocated to CP1.

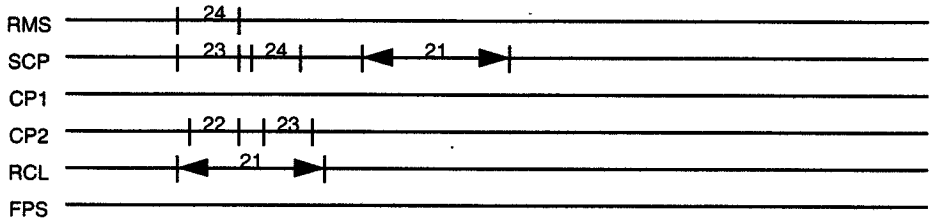


Figure 29 Frequently occurring deadlock because of absence of negotiation

5. Discussion

In this chapter, the design of the model is evaluated. Furthermore, the χ formalism is judged on its suitability for the design and evaluation of an agent based control system for the model factory.

5.1 The model of the control system

The model of the model factory's control system has been designed with the following objectives:

- the model has to resemble the model factory;
- the model's building blocks (i.e. the agents) have to be as generic as possible;
- the model has to reflect an agent based control system that scores well on the evaluation criteria, i.e. performance, robustness, and flexibility;
- the model has to be clear and transparent.

Resemblance of the model with the model factory
Resemblance with the real model factory has not been totally obtained. First and foremost, the χ model only takes a part of the model factory into account; the test and repair loop and the In-Process-Store are not considered (see Section 2.2).

A few differences between model and reality are in the component placement operations. A component placement workstation has space for two delivery locations times eight components. Only the first four components in a delivery location are accessible to the workstation. The component delivery is modelled as one delivery location of four components times three types. In addition, the component conveyors are able to transport multiple component trays at the same time, and even at the same position. A proper solution would be to model the transportation of trays one after the other, and thereby increasing the total transportation time. Finally, in the model the component placement operations do not actually need components. The Machine Controllers do not even have information about the type of operation they have to perform.

Furthermore, the operation times are fixed in order to obtain fixed schedules. In reality, they are random. To remedy this flaw, schedules of both the workstation that performs the operation and possible other workstations have to be corrected for the real operation times. The planning update message (type 11) may be used for this.

Generic building blocks
The workstation agents are kept as generic as possible. Communication among workstations via a network contributes to this. Without major modifications, new workstations can easily be added to the model. Situation-specific information such as the identification number of the last workstation in the system are known to Controllers by means of *xper* parameters. As a consequence of this genericity, the structure of a workstation agent is more complicated than strictly necessary. This structure is based on workstations that carry out the most complicated operations, namely the component placers. In principle, the remaining stations execute operations straight ahead, but now they possess more comprehensive processes that are not needed by stations such as Screen Printing and Reflow & Cleaning.

Evaluation criteria
The evaluation criteria as discussed in Section 1.5 are robustness, flexibility, and performance. The robustness of the agent based control system can not be evaluated, since disturbances are not modelled. However, it is expected that robustness in the agent based control system is increased compared to the distributed system without negotiation, due to the fact that routings were fixed in the latter system, whereas they are opportunistically 'composed' during operation in the agent based system. However, the effect is largely determined by the possibilities the manufacturing system offers. Here, the effect is only marginally, since in the present situation only the component placers can be interchanged to deal with malfunctions. If another station breaks down, the complete system will be blocked.

The flexibility, i.e. the modifiability of the agents, and the extensibility of the system, is better in the agent based system than in the previously implemented heterarchical control system (Timmermans, 1993b). Stations in the implemented heterarchical control system have knowledge about other stations. For instance, each station knows its direct 'neighbours'. If the factory is extended with a new workstation, the information its neighbours have of other stations needs to be updated. This is not necessary in the agent based system, since the agents communicate messages via the network. If a new agent is added to the system, the network is extended with a network interface that is connected to the new agent. Evidently, the switch component in the communication network of Figure 3 needs to be updated. Note that in the described heterarchical control model, stations do not have knowledge about each other. The Generator determines the routing, not the individual workstation.

However, it is hard to compare the two control systems in terms of modifiability. Workstation controllers in the previously implemented heterarchical control system have information about other controllers, which hampers extensibility as well as modifiability. However, the controllers in the agent based control system probably have more states and make more assumptions about the behaviour of other agents. This is not determined into detail (yet).

Alternative approaches for a broadcast throughout the system are available. The drawback of a broadcast to all stations is that an overload of message exchanges may paralyse the system. An alternative solution would be to apply audience restriction, for instance by giving the agents local knowledge of other agents' skills. For an example, the reader is referred to (Cantamessa, 1995).

Another possibility to realise audience restriction is to give intelligence to the network. In the model factory, workstations communicate with each other via a network. An intelligent Switch Element might transport messages to appropriate agents only rather than to all agents. This network construction could easily be extended into a broker. Then, agents report finished jobs (i.e. idle workstations) and operations to be executed to the broker, so the broker can match demand and supply of tasks.

The performance of the agent based control system is compared to distributed control systems without negotiation in the previous Chapter. The throughput times of the agent based system are only slightly better than the throughput times of the heterarchical system. Whereas the heterarchical system does not have routing flexibility at all, the possibilities of the agent based system to avoid a busy station and direct the batch to a less busy station are limited. This is caused by the absence of alternative workstations, except for placing components.

As compared to hierarchical control systems, studies show that the overall throughput times of agent based systems are worse than those of hierarchical control systems. After all, hierarchical control systems do not have the myopic view of agents; a hierarchical controller overlooks a

larger area than an individual controller and is capable of making less suboptimal decisions. An example of such a myopic view and resulting suboptimal decision is Figure 26 in the previous chapter, where the deadlock could have been prevented by a global system view.

The characteristics of the physical production system cause the agent based control system to perform only slightly 'better' or even worse than other control systems. This leads to the conclusion that the model factory is not a suitable production system for the application of an agent based control system, and brings up the question in what situations agent based systems truly make a difference.

The concept of the routing space is introduced to explain when agent based systems are most valuable and under which circumstances they can only have limited impact. The routing space is the set of possible transitions from one workstation to another, and is product specific. Possible transitions can be specified as ordered pairs of station of origin and destination. The more the routing space resembles a function, the less the agent based system is applicable. If the routing space is a function, at each station of origin a batch can only go to one station of destination. In this case, a deterministic schedule would suffice to exploit the 'full width' of the routing space, because there is no flexibility. That is, when a problem occurs, another station cannot be chosen. However, the more the number of ordered pairs in the routing space exceeds the number of process steps, i.e. from a station of origin a product can go to many stations of destination, the more the possibility to compose a schedule through negotiation at run time is going to be of value. In other words, agent based systems are more suitable in situations with many interchangeable workstations. In case of little uncertainty, however, a central scheduler would give more optimal routings and a better performance (Zwegers *et al.*, 1996).

In case the process plan is not fixed, the size of the routing space is also determined by another factor. The less a certain order between operations is required, the more transitions from one station to another are possible. The routing space will increase correspondingly. Control in an 'orderless' situation, in terms of process plans, requires extensive memory capabilities to keep track of batch history. Given the characteristics of agents, they are less suitable for such an orderless situation.

For the same reason of lack of memory capabilities, agent based systems cannot cope with situations in which it might be more favourable to group operations and have them performed at one machine. Agents have a quite myopic view, which might lead to suboptimal routings.

Model transparency
The transparency of the model is enlarged by distributing the various functions over various processes. Each process has its own task. If information is needed from other processes to fulfil the assigned task, the process communicates via clearly defined communication channels with other processes. The negotiation protocol is divided into various elementary tasks. It is not easy to obtain a good view of this protocol. Design changes in a process necessitate the reconsideration of the implementation of the negotiation protocol in other processes. Complications as a consequence of these changes require a good overview of the parallel processes.

5.2 The formalism χ

The formalism χ has been chosen for the simulation of an agent based control system of the model factory because of its flexibility and simplicity. The language itself enforces hardly any restrictions in the implementation of the models. However, in χ only fixed communication channels are possible, so that an agent can not be created but has to be 'statically' modelled in

the system. Dynamic execution of a job agent is therefore not possible (Coenen, 1995). Especially this restriction has lead to negotiation among workstation agents rather than among job agents and workstation agents.

The basic structure of χ is quite clear. The syntax and structure of the language are easy to learn. The mathematical background, however, is hard to see through. Constructions such as 'delta 0' may have their origin in this mathematical basis, but understanding their necessity requires probably more insight in the semantics of guarded command languages.

The tasks that have to be carried out are distributed over various processes, which simplifies the structure of the processes. However, this does not guarantee the transparency of the model. After all, modelling is an art (Rooda, 1996).

Modelling (in χ) necessitates to make thoughts explicit. These (often) simple thoughts might result in complex code. This could be caused by more complex presuppositions that form the basis of the thoughts, or because of inexperience of the modeller.

A particular strong feature of χ is the following construction:

```
*[ a ? l -> ...
 | b ? m -> ...
 | c ? n -> ...
 ]
```

This type of construction almost forces a process to return to the default state, as in Figure 30. That is, when a message arrives, a few statements are executed, and the process returns immediately to the default state, waiting for the next message to arrive through the channels *a, b,* or *c*. Returning immediately means that no communication, synchronisation, or delay takes place that might need time. Due to the asynchronous nature of the agent based control system, it is most unwise to execute a few commands and stay in some state; problems might occur when another message arrives.

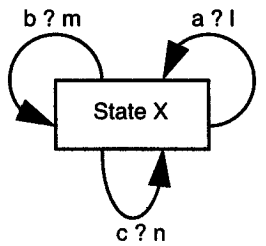


Figure 30 Returning to the default state

The formalism χ does not have a graphical user interface at this moment. Having such an interface might quicken the modelling time and might increase the overview of the model.

6. References

Brussel, H. Van. (1995). "Navigation" issues in intelligent autonomous systems. In: *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS '95)*, (U. Rembold, R. Dillmann, L.O. Hertzberger, and T. Kanade (Eds.)), pp. 42-52. IOS Press, Amsterdam.

Cantamessa, M. (1995). A few notes upon Agent-based Modelling of Manufacturing Systems. In: *Proceedings of the CIM at Work conference*, (J.C. Wortmann (Ed.)), pp. 301-317.

Chi. (1996). Example available at the Chi homepage. URL: http://www.tue.nl/wtb/wpa/se/chi/exam.htm

Coenen, F.W.J. (1995). A *Heterarchical Control structure for Flexible Production Systems* (in Dutch). M.Sc. thesis, Eindhoven University of Technology.

Jennings, N.R., P. Faratin, M.J. Johnson, P. O'Brien, and M.E. Wiegand. (1996). Using Intelligent Agents to Manage Business Processes. In: *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-agent Technology (PAAM96)*, pp. 345-360.

Mortel-Fronczak, J.M. van de, J.E. Rooda, and N.J.M. van den Nieuwelaar. (1995). Specification of a Flexible Manufacturing System Using Concurrent Programming. *Concurrent Engineering: Research and Applications*, Vol. 3, No. 3, pp. 187-194.

Rooda, J.E. (1996). *The Modelling of Industrial Systems*. Uncorrected preliminary version, lecture notes, Eindhoven University of Technology.

Smith, R.G. (1980). The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, Vol. C-29, No. 12, pp. 1104-1113.

Timmermans, P.J.M. (1993a). *Modular Design of Information Systems for Shop Floor Control.* PhD Thesis Eindhoven University of Technology.

Timmermans, P. (1993b). Control architectures and modular information systems: a comparative experiment. In: *Proceedings of the international conference on Advances in Production Management Systems (APMS '93)*, (I.A. Pappas and I.P. Tatsiopoulos (Eds.)), pp. 387-394. Elsevier Science Publishers.

Upton, D.M., M.M. Barash, and A.M. Matheson. (1991). Architectures and auctions in manufacturing. *International Journal of Computer Integrated Manufacturing*, Vol. 4, No. 1, pp. 23-33.

Wiendahl, H.-P., and V. Ahrens. (1995). Knowledge-Based Support for Planning and Control in Distributed Production Systems. In: *Proceedings of the IFIP 5.7 Working Conference on Managing Concurrent Manufacturing to Improve Industrial Performance*, pp. 429-443.

Zwegers, A.J.R., H.J. Pels, R.L.J. Schrijver, and R.J. van den Berg. (1996). An agent based control system for a model factory. In: *Proceedings of Advances of production management systems (APMS96)*, (N. Okino, H. Tamura, and S. Fujii (Eds.)), pp. 293-298.

Eindhoven University of Technology
Graduate School of Industrial Engineering and Management Science
Research Reports (EUT-Reports)

The following EUT-Reports can be obtained by writing to:
Eindhoven University of Technology, Library of Industrial Engineering and Management Science, Postbox 513, 5600 MB Eindhoven, Netherlands.
The costs are HFL 5.00 per delivery plus HFL 15.00 per EUT-Report (unless indicated otherwise), to be prepaid by a Eurocheque, or a giro-payment-card, or a transfer to bank account number 52.82.11.781 of Eindhoven University of Technology with reference to "Bibl.Bdk", or in cash at the counter in the Faculty Library.

20 LATEST EUT-REPORTS

EUT/BDK/88 Modelling of an agent based control systems for a model factory with the specification language Chi **Arian Zwegers, Raymond Schrijver, Angel Santana Alguacil**

EUT/BDK/87 Managing the strategic process : the impact of national/corporate culture on the strategic behavior of European MNC's **Rajesh Kumar, Jan Ulijn, Mathieu Weggeman, Robert van der Ven**

EUT/BDK/86 Dealing with risk : beyond gut feeling : an approach to risk management in software engineering **F.J. Heemstra, R.J. Kusters, R. Nijhuis, Th.M.J. van Rijn**

EUT/BDK/85 The development of an incident analysis tool for the medical field **W. van Vuuren, C.E. Shea & T.W. van der Schaaf**

EUT/BDK/84 Operations management and financial management information systems : a design approach for infinite and finite planning systems **P.E.A. Vandenbossche**

EUT/BDK/83 Gordian project : final report July 1996 **R.J. van den Berg, A.J.R. Zwegers**

EUT/BDK/82 Incidents in accident and emergency & anaesthesia **Wim van Vuuren**

EUT/BDK/81 Dada en adviseren geeft dadaviseren **Matthieu Weggeman**

EUT/BDK/80 Critical success factors in developing 'accepted control loops' **Harrie van Tuijl**

EUT/BDK/79 Organisatie-diagnose via de kwaliteitsincidenten methode **J.D. van der Bij, T.W. van der Schaaf, P.M. Bagchus**

EUT/BDK/78 Kwaliteitsmanagement in de gezondheidszorg : een onderzoek naar huidige ontwikkeling en onderzoeksbehoeften in ziekenhuizen **T. Vollmar en J.D. van der Bij**

EUT/BDK/77 Het ene artikel is het andere niet! : een onderzoek naar de problemen omtrent de slechte afstemming tussen artikelstamgegevens in de levensmiddelenbranche **B. Vermeer**

EUT/BDK/76 Wegtransport : vitaal voor economie, welvaart en welzijn **J.P.M. Wouters e.a.**

EUT/BDK/75 Diagnosing the production organisation of SMES **M.J. Verweij**

EUT/BDK/74 Describing, analysing and designing with the production description language **M.J. Verweij**

EUT/BDK/73 Purchasing's development role : the internal and external integration of purchasing in technological development processes : intermediate report I **J.Y.F. Wynstra**

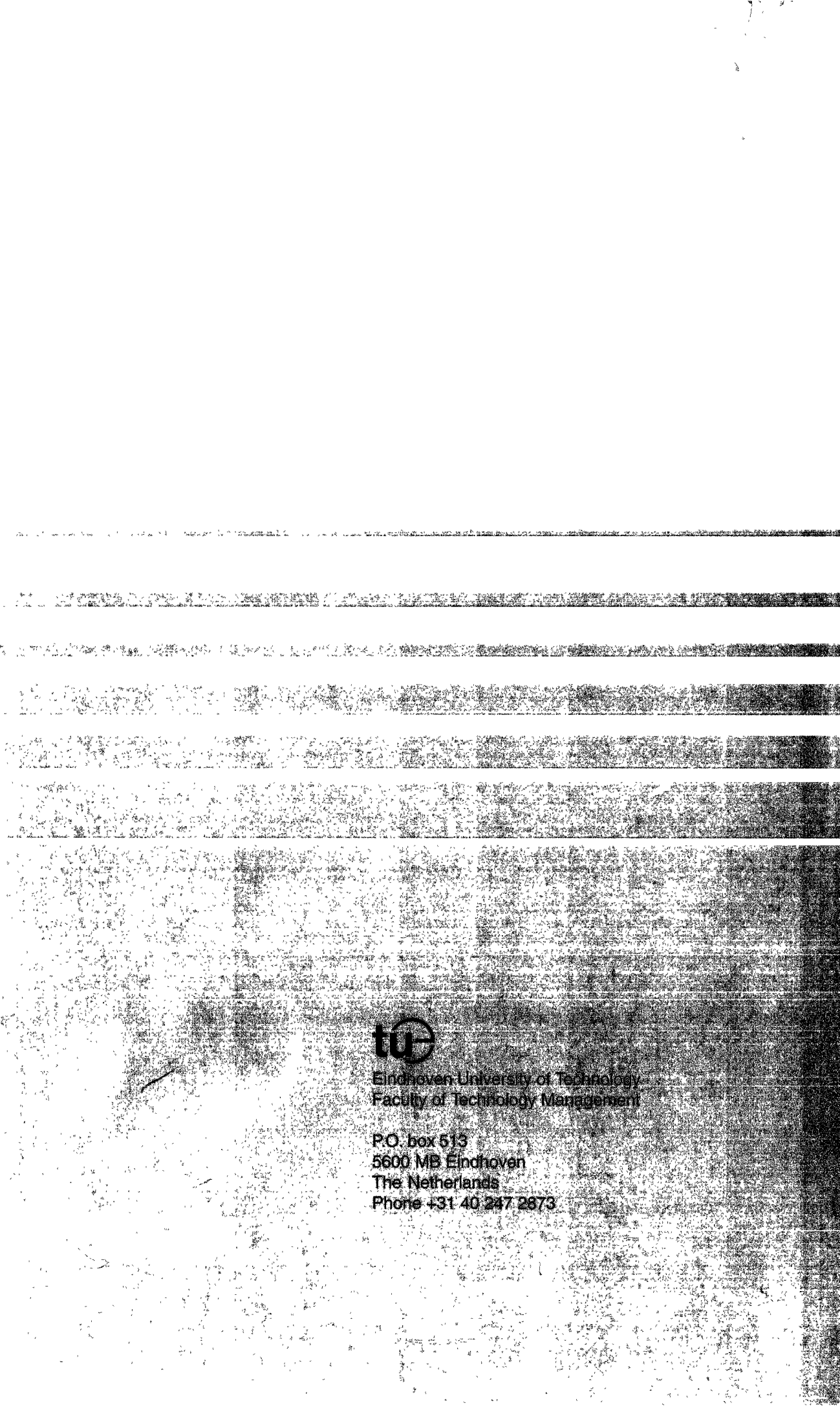
EUT/BDK/72 De problemen van hergebruik gezien vanuit de stofstromenproblematiek **A.J.D. Lambert**

EUT/BDK/71 Problemen en knelpunten bij gebruik van MRP in de praktijk : onderzoeksrapport **M.J. Euwe**

EUT/BDK/70 De groothandel is dood. Leve de groothandel! : een branchegericht onderzoek naar de toekomst van de groothandel en de rol van informatie technologie **M.J. Euwe**

EUT/BDK/69 Methodologies for information systems investment evaluation at the proposal stage : a comparative review

Th.J.W. Renkema, E.W. Berghout



Eindhoven University of Technology
Faculty of Technology Management

P.O. box 513
5600 MB Eindhoven
The Netherlands
Phone +31 40 247 2873