# Specification of an Automatic Manufacturing System: A Case Study in Using Integrated Formal Methods*

Heike Wehrheim

Universität Oldenburg
Fachbereich Informatik
Postfach 2503, D–26111 Oldenburg, Germany
wehrheim@informatik.uni-oldenburg.de

**Abstract.** An automatic manufacturing system serves as a case study for the applicability of an *integrated formal method* to the specification of software systems. The formal method chosen is *CSP-OZ*, an integration of the state-oriented formalism Object-Z with the process algebra CSP. The practicability as well as limitations of CSP-OZ are studied. We furthermore employ a graphical notation (class diagrams) from the Unified Modelling Language to describe the *architectural view* of the system. The correctness of the obtained specification is checked by a translation into the input language of the CSP model checker FDR and a following property check.

## 1 Introduction

Recently, there is an emerging interest in formal methods which combine specification techniques for different views on systems. In particular, methods integrating static aspects (data) and dynamic aspects (behaviour) are investigated (see for example [Obj99, Que96, GP93, TA97, GS97, Smi97, MD98]). The advantage of these methods is that different views on a system can be conveniently modelled. Integrated methods are in particular important for the specification of *software* for reactive systems, which have to cope with a number of different aspects of systems: large data descriptions, dynamic behaviour, timing constraints and analog components. In this paper, we investigate the usefulness and applicability of an integrated formal method to the specification of software for manufacturing systems. The formalism chosen is CSP-OZ [Fis97], a combination of the process algebra CSP [Hoa85] and an object-oriented extension of Z, Object-Z [DRS95]. CSP-OZ integrates a state-oriented with a behaviour-oriented view and thus allows to specify different aspects of a system within a single formalism. A semantics for the combined formalism has been given in [Fis97].

* This work was partially funded by the Leibniz Programme of the German Research Council (DFG) under grant Ol 98/1-1.

T. Maibaum (Ed.): FASE 2000, LNCS 1783, pp. 334–348, 2000.

Our case study is the specification of a *holonic manufacturing system*. This case study is part of the german research program "Integration of Software Specification Techniques"[1]. The term "holon" has its origin in the greek word "holos" (whole) and describes an autonomous, flexible agent. The term "holonic manufacturing systems" refers to systems where the transportation of material within the plant are managed by holonic transportation systems, i.e. transportation systems without drivers and without central scheduling device [WHS94]. The purpose of the manufacturing system is the processing of workpieces by different machine tools. Two stores in the plant serve as containers for workpieces. The holonic transportation agents are responsible for transportation of workpieces between machine tools and stores. The throughput of workpieces in the plant should be as high as possible. An elaborate communication protocol between machine tools, stores and transportation agents ensures that the agent with the smallest cost for transportation is used for a particular transportation job.

The CSP-OZ specification models an abstract view of the system, considering the *activities* of machine tools and agents as a modelling entity, i.e. we do not specify the manufacturing system on machine level describing movements of roboter arms etc., but rather have concentrated on the software used in the different components. A particular focus was laid on the specification of the communication scheme between components and the influence of the data part thereon. The specification clearly showed the strength of CSP-OZ as a specification formalism for software systems but also revealled some weaknesses, for instance the lack of specifying timing constraints.

An additional aim of the work presented here was to investigate the possibility of integrating more informal graphical notations, used in industry, with a formal method. As a first approach, we have chosen to use the class diagrams of UML for the specification of the communication structure of the system. This turned out to fit well to the object-oriented method CSP-OZ; thedifference to an ordinary use of class diagrams lies in the interpretation of associations in the context of distributed communicating systems: since objects may be (and in our case are) physically distributed, all interactions among active objects have to be interpreted as *communications* (in the sense of message exchanges via channels).

An advantage of using formal methods in the specification of software is their precise formal semantics. This advantage can best be exploited when some property checking on the specification can also be performed. Using a technique proposed in [FW99] – translating CSP-OZ specifications into the input language of the CSP model checker FDR [FDR97] – we carry out some correctness checks on the specification, most notably a check for *deadlock-freedom*. Given the complex communication protocol among machine tools and transportation agents, deadlock-freedom is not trivial to achieve.

In Section 2 we start with a brief introduction of our specification formalism CSP-OZ. Section 3 presents (part of) the specification of the manufacturing system, and Section 4 describes the property check on the specification.

[1] http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/index.html.

## 2 CSP-OZ

CSP-OZ is an *object-oriented formal method* combining Object-Z [DRS95] (an object-oriented extension of Z) with the process algebra CSP. The general idea is to augment the state-oriented Object-Z specification with the specification of behaviour in the style of CSP. A CSP-OZ specification describes a system as a collection of interacting objects, each of which has a prescribed structure and behaviour. Communication takes place via *channels* in the style of CSP. CSP-OZ has a formal semantics on the basis of CSPs failure-divergence model [Fis97].

In general, a CSP-OZ specification consists of a number of paragraphs, introducing classes, global variables, functions and types. Instances of the classes can be combined via CSP composition operators. Here, we will briefly describe the form of class specifications and illustrate them by a small example of a one-place buffer. A CSP-OZ class has the following basic structure:

*Name*(*formal parameters*)

| | |
|---|---|
| `inherit` *superclass* | [inherited superclasses] |
| `chan` *channelname* | [channel definitions] |
| `main` = ... | [CSP-Part] |
| *type and constant definitions*<br>*state schema*, *initial state schema*<br>*operation schemas* | [Z-Part] |

The state schema gives the *attributes* of a class, the operation schemas their *methods*. The initial state schema fixes the initial values of attributes. For every method which can be called on an instance of the class, a corresponding channel has to be defined. In the operation schemas methods are defined by *enable* and *effect predicates*, which give the enabling conditions for an application of an method and its effect on the attributes and communicated values. The parameters of a method can be of type *input* (denoted $x?$), *output* ($x!$) or *simple* (solely $x$). A simple parameter is one on which both communication partners agree, there is no direction in the flow of communication value. Simple parameters can for instance be used for *object references*: in a distributed setting, a method call $m$ to a particular object $O$ (usually written as $O.m$, where $O$ is a reference to the object) can now be expressed, using communication via channels, as $m.O$, where $m$ is the name of a channel and $O$ an instantiation of a simple parameter.

Every class may *inherit* attributes and methods from one or more other classes. A class may have a number of formal parameters, which have to be replaced by actual parameters when the class is instantiated. In the CSP-Part of the specification the behaviour of a class is defined, i.e. the order of execution of methods is fixed (sometimes also called the *synchronisation constraint* of the class). The CSP-Part describes the *data-independent* part of behaviour, it may not refer to attributes of the class (therefore all parameters of channel names occur as input parameters, $chan?x$). The data-dependent aspects are encoded in the enabling conditions of the methods. Thereby a clean separation of data and behaviour aspects is obtained.

Below a CSP-OZ specification of a class *Buffer* is given. The basic type of elements in the buffer is *Element*.

$[Element]$

**Buffer**

$\texttt{chan}\ put : [\, elem? : Element \,]$
$\texttt{chan}\ get : [\, elem! : Element \,]$
$\texttt{main} = put?x \rightarrow get?x \rightarrow \texttt{main}$

$contents : \mathbb{P}\, Element$

$\#contents \leq 1$

**Init**

$contents = \varnothing$

**enable_*put***

$contents = \varnothing$

**effect_*put***

$\Delta(contents)$
$elem? : Element$

$contents' = contents \cup \{elem\}$

**enable_*get***

$\#contents = 1$

**effect_*get***

$\Delta(contents)$
$elem! : Element$

$elem \in contents$
$contents' = contents \setminus \{elem\}$

The class has one attribute *contents* (specified in the state schema) and methods *put* and *get*. A primed attribute name stands for the value of the attribute after the execution of the method. The $\Delta$-declaration declares the attributes which may be changed by the method. The CSP-Part specifies that *put* and *get* methods may only be used alternately. This is also guaranteed by the enabling conditions of the methods, i.e. we could as well either leave out the CSP-Part or set the enabling conditions of methods to true without changing the behaviour of class *Buffer*.

### 2.1 CSP Operators

In this example, we have used just one CSP operator, the prefix operator $\rightarrow$. In the specification of the manufacturing system, a number of other operators will be used, which are now briefly explained.

- ; denotes sequential composition of processes;
- $|||$ denotes parallel composition with no synchronisation, $\|_A$ is parallel composition with synchronisation on all events in the set $A$, i.e. the components of a parallel composition have to jointly execute events in $A$;

- $\setminus A$ is hiding of events in the set $A$;
- $\Box$ is an external choice; external means that the choice can be influenced by an environment requesting certain events;
- $[name1 \leftarrow name2]$ is a renaming of $name1$ into $name2$.

Furthermore, most operators may be used in a replicated version, e.g. $||| \ a : A \bullet P(a)$ is a parallel composition of all processes $P(a)$, where $a$ is in $A$.

A last operator is to be explained, which will be used quite often: the CSP *timeout* operator. The intention of this operator is to abstractly model a timeout in the absense of a notion of time in CSP. The process $P \triangleright Q$ has the following behaviour: with any visible action of $P$, the choice between $P$ and $Q$ is decided, if no action from $P$ happens, the process times out and behaves like $Q$.

A more thorough introduction to CSP can be found in [Hoa85, Ros97], an introduction to Z in [Spi92], to Object-Z in [Smi00].

## 3 Specification

Next, we present the specification of the automatic manufacturing system. This case study is part of the german DFG priority program "Integration of specification techniques with applications in engineering". The automatic manufacturing system consists of the following parts: two stores (*In* and *Out*), one for workpieces to be processed (the in-store), one for the finished workpieces (the out-store), a number of holonic transportation systems (hts) ($T1, T2, \ldots$) and three machine tools (short: wzm for german "Werkzeugmaschinen") $A$, $B$ and $C$ for processing the workpieces. Every workpiece has to be processed by all three machine tools in a fixed order ($In \rightarrow A \rightarrow B \rightarrow C \rightarrow Out$). The hts' are responsible for transportation of workpieces between machine tools and stores. The hts' work as autonomous agents, free to decide which machine tool to serve (within some chosen strategy). Initially the in-store is full and the out-store as well as all wzm are empty. When a wzm is empty or contains an already processed workpiece it broadcasts a request to the hts in order to receive a new workpiece or to deliver one. The hts' (when listening) send some offer to the wzm, telling them their cost for satisfying the request. Upon receipt of offers the wzm decides for the best offer and give this hts the order, which then executes it. This way, all workpieces should be processed by all three tools and transported from the into the out-store.

The specification language we employ is an *object-oriented* formal method. The specification thus consists of a number of classes, somehow related to one another. To facilitate the understanding of the overall structure of the specification, we describe it by means of a *class diagram* (in the style of the Unified Modelling Language, UML). Class diagrams are (so far) not part of the formal method CSP-OZ, we just use them as a graphical means for showing the structure of specifications. Class diagrams show the static structure of the model, in particular, the things that exist, their internal structure and their relationships to other things. In particular, we find: boxes, denoting *classes*; arrows, describing *generalisation* (inheritance); simple arcs, standing for *associations* and arcs

with filled diamonds at the end, denoting *compositions*. An association shows a possible interaction between two classes. A composition is a strong form of an association: it describes a relationship between a "whole" and its parts, where the existence of the parts depends on the whole. Class diagrams are traditionally used for data modelling. In this paper we take a different view on class diagrams. All classes in the diagram are assumed to be *active*. When using class diagrams in the description of distributed communicating systems, active objects most often reside on different locations, hence any interaction has to take the form of a *communication*. Thus associations and compositions stand for some particular form of composition (now in the process algebra sense) of the classes, using communication via channels for interaction. Associations are parallel compositions with communication via the channels which are the *names* of the association. Composition is stronger: the class and its components are combined in parallel but now all channels between the class and its components are hidden to the outside.

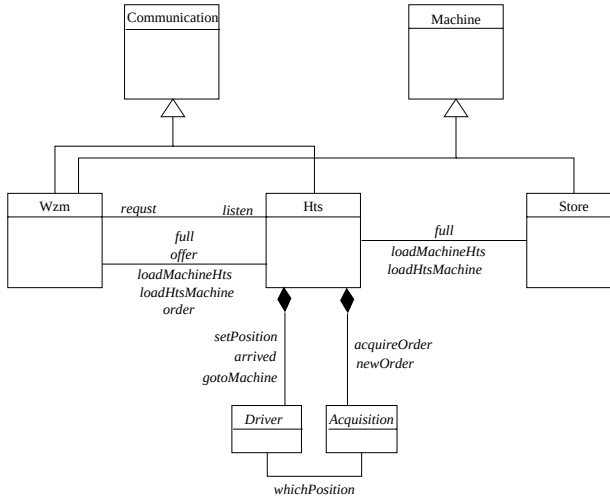



**Fig. 1.** Class diagram for manufacturing system

In Figure 1 the class diagram for the manufacturing system is given. The main classes of our specification are *Wzm*, *Hts* and *Store*. The class diagram furthermore contains two superclasses from which these classes are derived: a superclass *Machine* specifying the basic behaviour of stores and machine tools (with attributes like contents and methods like loading), and a superclass *Communication* defining a protocol for broadcast communication. The class *Hts* embodies two other classes: one for managing the driving in the plant hall (*Driver*) and a class for acquisition of new orders (*Acquisition*). The use of composition and not association here implies that every *Hts* inevitably has components *Driver* and *Acquisition*.

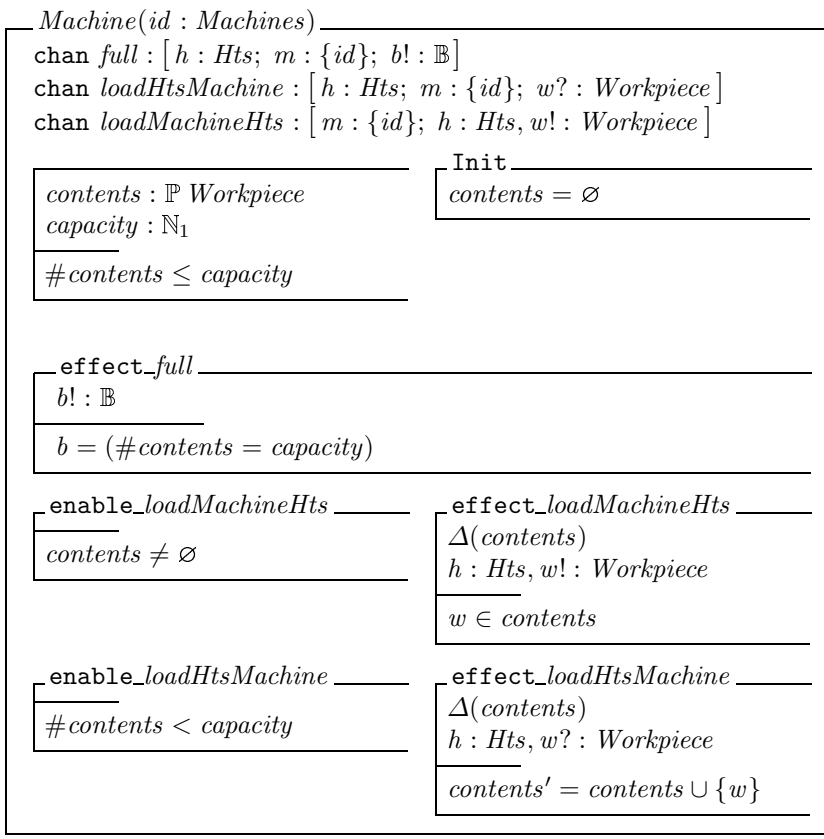
**Class specifications.** The classes themselves are now specified with CSP-OZ. At the end we will see how they can be combined by CSP operators in order

to achieve the above depicted structure. We start with the definition of a basic type and some abbreviations: *Workpiece* is the basic type for workpieces, *Wzm* specifies the set of all machine tools, *Hts* the holonic transportation systems, and *Store* the stores.

$$[Workpiece] \qquad Wzm == \{A, B, C\}$$
$$Hts == \{T_1, T_2\} \qquad Store == \{In, Out\}$$
$$Machines == Stores \cup Wzm$$

The type *Coord* is used to describe the position of machines and agents in the hall. *Status* describes the status of workpieces in machines (processed or not yet processed): $Coord == \mathbb{N} \times \mathbb{N}$, $Status == \{finished, notFinished\}$.

We start with the specification of the superclass *Machine*. It should capture the basic common properties of machine tools and stores: they may contain a number of workpieces (limited by some capacity), they may load and deload workpieces, and tell others whether they are empty or full.




The first part of the specification describes the basic interface between machines and hts. Afterwards the attributes of the class are defined and the initial state values are given. The last part gives the enabling conditions and effects for the

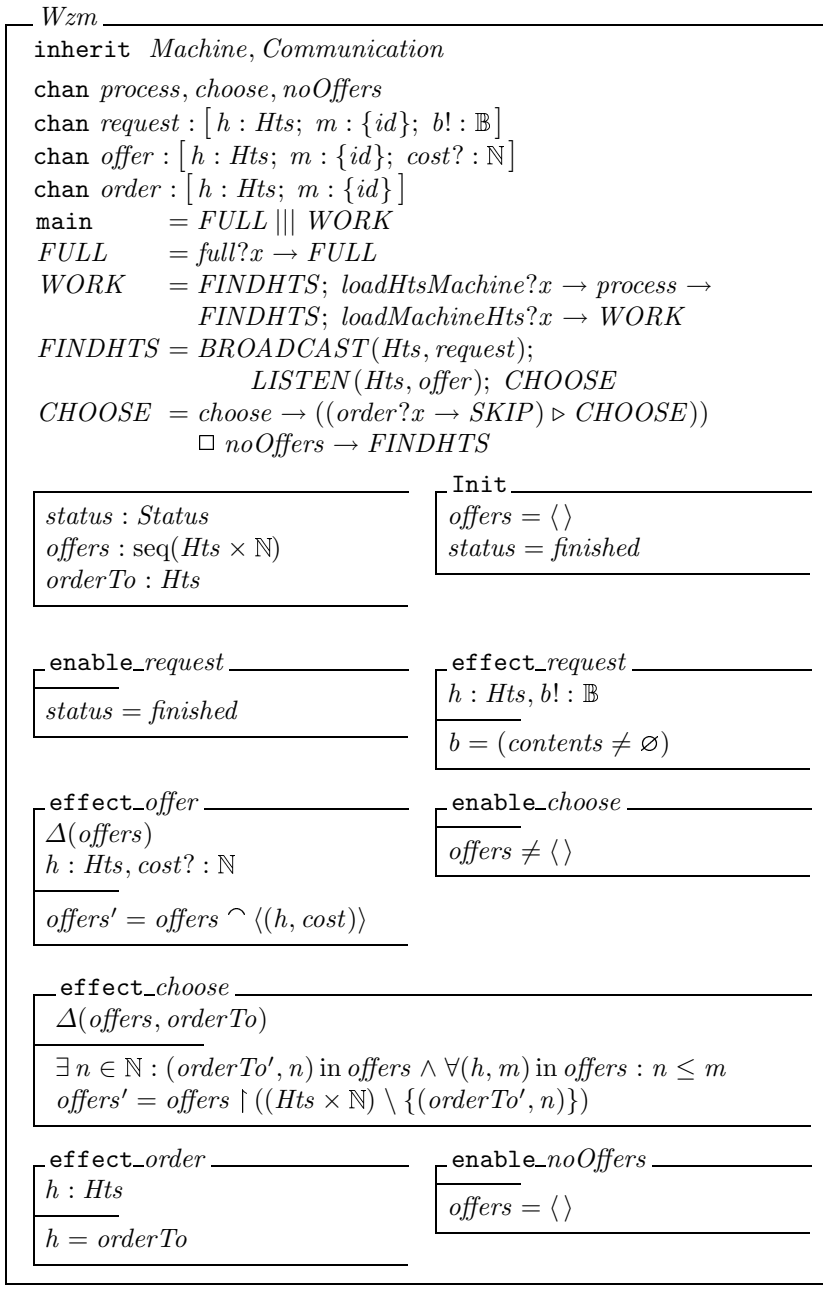
execution of operations (e.g. only load workpieces from a machine to an hts when the machine is not empty). Since we just define the basic ingredients of machines here, we have no CSP behaviour descriptions.

The superclass *Communication* provides two protocols for broadcast communication between agents and machine tools. Here is one point where we reach the limits of CSP-OZ: CSP with its *synchronous* communication paradigm cannot exactly describe the radio communication between tools and agents. Radio communication is somehow a mixture of synchronous and asynchronous communication: the sender may always send its message (asynchronously), the receiver can only receive the message when it listens at exactly the time the message is sent (synchronously). Since we have no notion of time in our specification language, we model this type of communication with the CSP timeout operator: the sender tries a synchronous communication with every receiver; when this fails it may timeout. Similarly the receiver can timeout when no communication is possible. This behaviour is modelled by the processes $BROADCAST$ and $LISTEN$, which have the set of receivers ($to$), senders ($from$) respectively, and the communication channel ($comm$) as parameters.

*Communication*

$$BROADCAST(to, comm) =$$
$$\quad ||| \; rec : to \bullet ((comm?x.rec?y \rightarrow SKIP) \triangleright SKIP)$$
$$LISTEN(from, comm) =$$
$$\quad ||| \; sender : from \bullet ((comm!from?x?y \rightarrow SKIP) \triangleright SKIP$$

The class *Wzm* specifying machine tools is a subclass of both *Machine* and *Communication* (i.e. we have multiple inheritance here). Inheritance is semantically the *conjunction* of the Z part of the superclass with the Z part of the subclass, and *parallel composition* of the CSP parts with synchronisation on all events that occur in both CSP specifications. The process equations of the superclass are inherited by the subclass and may be used by them (code re-use). This is for instance the case for the protocols of superclass *Communication*: inheritance makes the two process names accessible within the subclass *Wzm*. The basic behaviour of a wzm is as follows: it has to find an hts fetching some workpiece for it, load a workpiece, process it, find another hts to take it over, deload it and start again from the beginning. Additionally it always has to be able to tell others wether it is full or empty. This behaviour is specified by the CSP process of class *Wzm*. The search for an hts proceeds as follows: the wzm broadcasts a request to all hts (telling them whether it is full or empty), the hts that have listened may send a reply offering some cost for the transport. The wzm then chooses the offer with the smallest cost and tries to order this hts. If the ordering does not succeed (the hts may have decided for another job), they choose another offer until either the order succeeds or no further offers are available upon which the wzm makes its request again.




The basic behaviour of wzms is specified by the CSP process `main`, all data dependent aspects are specified within the CSP part (for instance the choice for the offer with the smallest cost is encoded in the effect schema of operation *choose*). This allows for a clear separation of static and dynamic aspects, and

also allows for an easy change (for instance when other criteria should be used for choosing an offer).

The class *Hts* describes the principle behaviour of a holonic transportation agent. The driving and order acquisition specific aspects are specified within the components *Driver* and *Acquisition*. Since the most interesting part of an hts (negotiation with wzms) is specified within the component *Acquisition* we refrain from giving the specification of classes *Hts* and *Driver* here and instead concentrate on *Acquisition*. The main task of class *Acquisition* is the negotiation of new orders. Initially it waits for a call from class *Hts* to acquire a new order. It asks for its current position and listens to the requests made by the wzm. All received requests have to be checked whether they can be satisfied, for instance, if the requesting wzm is full, it has to be checked whether the wzm, next to the requesting one, is empty (in order to asure that a workpiece can also be delivered). This check is encoded in the effect of operation *full*: all wzm are asked whether they are full ($ASK$) and the set of current requests is modified according to the answer. From all satisfiable requests the one with the least cost (distance from current position to wzm) is chosen and an offer is made to the requesting wzm. If this offer succeeds and an order is accomplished, class *Acquisition* informs the *Hts* about this. If the offer does not succeed, the set of current requests is emptied and finding a new order starts again.

*Acquisition*(*master* : *Hts*)

inherit *Broadcast*

chan *newOrder* : $[\, ord! : Wzm \times \mathbb{B} \,]$
chan *whichPosition* : $[\, pos? : Coord \,]$
chan *listen* : $[\, h : \{master\};\ wzm : Wzm;\ b? : \mathbb{B} \,]$
chan *offer* : $[\, h : \{master\};\ wzm : Wzm;\ cost! : \mathbb{N} \,]$
chan *order* : $[\, h : \{master\};\ wzm : Wzm \,]$
chan *requestsThere*, *noRequests*, *emptyRequests*, *acquireOrder*

$\mathtt{main} = acquireOrder \rightarrow whichPosition?x \rightarrow FINDORDER$
$FINDORDER = LISTEN(Wzm, listen);$
$\quad (requestsThere \rightarrow ASK;$
$\quad\quad ((offer?x \rightarrow ((order?x \rightarrow newOrder?x \rightarrow \mathtt{main})$
$\quad\quad\quad \triangleright\ emptyRequests \rightarrow FINDORDER))$
$\quad\quad\ \triangleright\ emptyRequests \rightarrow FINDORDER)$
$\quad\quad \Box\ (noRequests \rightarrow FINDORDER))$
$\quad \Box\ (noRequests \rightarrow FINDORDER)$
$ASK = ||| ma : Machines \bullet (full?x!ma?y \rightarrow SKIP)$

$currRequests : \mathbb{P}(Wzm \times \mathbb{B})$
$orderFrom : Wzm$
$position : Coord$

**Init**
$currRequests = \varnothing$

**effect_*newOrder***

$\Delta(currOffers)$
$ord! : Wzm \times \mathbb{B}$

---

$first(ord) = orderFrom, ord \in currRequests$
$currOffers' = \varnothing$

**effect_*whichPosition***

$\Delta()$
$pos? : Coord$

---

$pos = position$

**effect_*emptyRequests***

$\Delta(currRequests)$

---

$currRequests' = \varnothing$

**enable_*noRequests***

---

$currRequests = \varnothing$

**enable_*requestsThere***

---

$currRequests \neq \varnothing$

**effect_*listen***

$\Delta(currRequests)$
$wzm : Wzm, b? : \mathbb{B}$

---

$currRequests' = currRequests \cup \{(wzm, b)\}$

**enable_*offer***

---

$currRequests \neq \varnothing$

**effect_*order***

$\Delta(orderFrom)$
$wzm : Wzm$

---

$orderFrom' = wzm$

**effect_*offer***

$\Delta()$
$wzm : Wzm, cost! : \mathbb{N}$

---

$\exists\, b \in \mathbb{B} : (wzm, b) \in currRequests \wedge$
$\forall (m, q) \in currRequests \bullet$
$\quad dist(position, target(wzm, b)) \leq dist(position, target(m, q))$
[function target determines the position of the target of the request]
$cost = dist(position, place(wzm))$

**effect_*full***

$\Delta(currRequests)$
$wzm : Wzm, b? : \mathbb{B}$

---

$currRequests' =$
$\quad currRequests \setminus \{(w, f) \bullet (if\ f\ then\ next(w) = wzm \wedge b) \vee$
$\qquad\qquad\qquad\qquad (if\ \neg f\ then\ prev(w) = wzm \wedge \neg\, b)\}$

This completes the part of the class specifications. Due to lack of space we leave out the specifications of the other classes.

**System specification.** The classes now have to be instantiated and the created objects have to be combined to give the automatic manufacturing system. First we define a process *compoHts* describing the composition of *Hts* with *Driver* and *Acquisition*. Composition in class diagrams is translated into a parallel composition of components, where in contrast to ordinary association, the synchronised events are hidden afterwards. The synchronisation set is derived from the inscriptions of the association arcs in the class diagram: *Driver* and *Acquisition* synchronise on the set $A = \{whichPosition\}$, *Hts* with *Driver* and *Acquisition* on $B = \{setPosition, setId, arrived, gotoMachine, newOrder, acquireOrder\}$. We also carry out the renaming here which is induced by the association between *Hts* and *Wzm* in which the participants of the association have different roles (channel names *request* and *offer* at the ends of the association). However, we could also have chosen to use a linked parallel composition between *Hts* and *Wzm*.

$$compoHTS(id) = \\ (Hts(id) \parallel_B (Driver(id) \parallel_A Acquisition(id))) \setminus (A \cup B)[listen \leftarrow request])$$

The automatic manufacturing system is then obtained as a parallel composition of an appropriate number of class instances. In the choice of synchronisation sets we have to be a little more careful now: some channel names (e.g. *loadHtsMachine*) occur on more than one association; since the associations are however not ternary, this is not ment to be a synchronisation of three or more objects. In this case synchronisation now has to be distinguished by parameters and not by channel names alone. All parameters used for this purpose should be *simple* and stand for the identities of the involved objects. As an example: $Store(In)$ and all hts synchronise on $loadMachineHts.In$, whereas $Wzm(A)$ and the hts synchronise on $loadMachineHts.A$.

$$AutoManuSystem = \begin{array}{c} Store(In) \\ \parallel_C \\ (|||\ hts : Hts \bullet compoHTS(hts)) \\ \parallel_D \\ (|||\ wzm : Wzm \bullet Wzm(wzm)) \\ \parallel_E \\ Store(Out) \end{array}$$

where we have the following synchronisation sets:

$$C = \{loadMachineHts.In, full.T1.In, full.T2.In\},$$
$$D = \{loadMachineHts.A, loadMachineHts.B, loadMachineHts.C,$$
$$loadHtsMachine.T1.A, loadHtsMachine.T1.B, loadHtsMachine.T1.C,$$
$$loadHtsMachine.T2.A, loadHtsMachine.T2.B, loadHtsMachine.T2.C,$$

$$full.T1.A, full.T1.B, full.T1.C,\ full.T2.A, full.T2.B, full.T2.C,$$
$$request, offer, order\},$$
$$E = \{loadHtsMachine.T1.Out, loadHtsMachine.T2.Out,$$
$$full.T1.Out, full.T2.Out\}.$$

This completes the specification of the automatic manufacturing system. So far, this models the manufacturing system on a rather abstract level; we have for instance not modelled how the loading and deloading of workpieces is handled on the actual machine level. Nevertheless, the specification contains all activities performed by the system and describes the communication scheme for interaction. The communication scheme of a system is the major source for errors leading to deadlocks of the system. In the next section, we describe how we can prove deadlock freedom of our specification.

## 4 Verification

The verification of the manufacturing system follows ideas proposed in [FW99] (building on ideas of [MS98]): the CSP-OZ specification is translated into the CSP dialect of the model-checker FDR [FDR97], which can then be used to verify properties on the specification. The formal basis for this translation is the failure-divergence semantics of CSP-OZ classes. The CSP dialect of FDR is a combination of CSP with a functional language. The functional part of FDR-CSP can be used for modelling the Z-part of the specification. The translation cannot handle CSP-OZ completely, but a rather large portion of it. The translation for instance requires instantiation of basic types and restriction of variables to finite domains; however we do not have to eliminate nondeterminism from the specification, which usually has to be done when "executing" a formal specification.

For the manufacturing system, we thus have to choose some concrete set of values for the basic type *Workpiece* and we have to restrict *Coord* to a finite space (say $1 \ldots 10 \times 1 \ldots 10$). The most severe restriction, necessary for model-checking, concerns the initial values of the stores. We have not given the specification of the stores here, but of course the in-store has to be filled with a certain number of workpieces. Ideally, the model-checker should be able to verify deadlock-freedom of the specification for *any* number of workpieces in the in-store. However, this is beyond the range of model checkers like FDR. We thus have to perform the model-checking for a fixed number of workpieces in the store.

Having chosen these concrete values, the specification can be translated into FDR-CSP[2]. Two properties have been checked on the translated specification:

1. Deadlock-freedom and
2. adherence to the correct ordering of processing.

[2] At the moment the translation has to be done manually, but an implementation is under development.

By the second point, we mean that every workpiece has to be processed by the machine tools in the correct order. The second property was verified by *hiding* all events besides the event *process* and only observing the ordering of processing workpieces. This shows that every workpiece of the in-store is correctly carried to the wzm and processed in the right order. As an example for the performance of FDR on the case study: the labelled transition system for an instantiation with two hts and three wzm has 4213677 states and the second property can be checked in 1008 seconds cpu time on a SPARC Ultra.

The verification detected three errors in the first specification: an incorrect setting of initial values of one class, a wrong order of events in the CSP-part of a class and a wrong synchronisation set in the parallel composition of classes. So, although we cannot claim to have verified correctness of the specification for all possible instantiations of the system, we have been able to use a model-checker to find general errors.

## 5 Conclusion

In this paper we investigated the applicability of an integrated formal method to the specification of an industrial-scale software system. The case study clearly showed the advantages of using CSP-OZ, in particular the need for a formalism combining behaviour and data specification, but also revealed some drawbacks, for instance the inability of expressing timing requirements. The later aspect would especially be important when evaluating the performance (throughput in time) of the manifacturing system.

Besides designing a specification, we were also able to prove correctness properties of the design by means of a translation into the input language of the model checker FDR. The model checking process however always relies on fixing a particular instantiation of the system.

The case study also demonstrated the usefulness of employing graphical modelling languages in the design of the specification. We intend to further extend the possibilities of using object-oriented design methods together with CSP-OZ, especially the UML profile UML-RT [SR98], which seems to be well suited for the description of distributed communicating systems.

## References

[DRS95] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[FDR97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, Oct 1997.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[FW99] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods (IFM)*, pages 315–334. Springer, 1999.

[GP93] J.F. Groote and A. Ponse. Proof theory for $\mu$-CRL: A language for processes with data. In *Semantics of specification languages*, Workshops in Computing. Springer, 1993.

[GS97] A. J. Galloway and W. Stoddart. An operational semantics for ZCCS. In M. Hinchey and Shaoying Liu, editors, *Int. Conf. of Formal Engineering Methods (ICFEM)*. IEEE, 1997.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[MD98] B. P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, April 1998.

[MS98] A. Mota and A. Sampaio. Model-checking CSP-Z. In *Proceedings of the European Joint Conference on Theory and Practice of Software*, volume 1382 of *LNCS*, pages 205–220, 1998.

[Obj99] Object Management Group. *OMG Unified Modeling Language Specification*, June 1999. version 1.3.

[Que96] J. Quemada, editor. *Revised working draft on enhancements to LOTOS (V4)*. ISO, 1996.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[Smi97] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Proceedings of FME 1997*, volume 1313 of *LNCS*, pages 62–81. Springer, 1997.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computer Science, 2nd edition, 1992.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime, 1998.

[TA97] K. Taguchi and K. Araki. Specifying concurrent systems by Z + CCS. In *International Symposium on Future Software Technology (ISFST)*, pages 101–108, 1997.

[WHS94] E. Westkämper, M. Höpf, and C. Schaeffer. Holonic manufacturing systems. In Lake Tahoe HMS Consortium, editor, *Holonic manufacturing systems*, 1994.