Linking CSP-OZ with UML and Java: A Case Study[⋆]

Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim

Department of Computing Science
University of Oldenburg
26111 Oldenburg, Germany

{michael.moeller,olderog,rasch,wehrheim}@informatik.uni-oldenburg.de

Abstract. We describe how CSP-OZ, an integrated formal method combining the process algebra CSP with the specification language Object-Z, can be linked to standard software engineering languages, viz. UML and Java. Our aim is to generate a significant part of the CSP-OZ specification from an initially developed UML model using a UML profile for CSP-OZ, and afterwards transform the formal specification into assertions written in the Java Modelling Language JML complemented by CSP_{jassda}. The intermediate CSP-OZ specification serves to verify correctness of the UML model, and the assertions control at runtime the adherence of a Java implementation to these formal requirements. We explain this approach using the case study of a "holonic manufacturing system" in which coordination of transportation and processing is distributed among stores, machine tools and agents without central control.

Keywords. CSP, Object-Z, UML, Java, assertions, runtime checking

1 Introduction

Object-oriented (OO) design and programming languages play a major role in software development. The Unified Modeling Language (UML) [38,33] is an industrially accepted OO-modelling and design language; Java [15] is a widely used modern OO-programming language. UML and Java are thus likely to be used together in an object-oriented system development.

While UML and Java are well suited for modelling and implementation, they fall back as far as *correctness* issues are concerned. One of the main criticisms against UML is its lack of precision; Java programs are difficult to formally analyse even for experts [1,21]. Hence, with respect to reliability, a UML and Java based software development would gain from being complemented with a *formal* approach to system design. Since UML with its various diagram types allows for a multi-view modelling of a system, a formal method with the ability of specifying different aspects of a system as well is needed here. For this purpose we took CSP-OZ [12], an *integrated* formal method combining a state-based specification language (Object-Z [9]) with a behaviour-oriented language (CSP

[⋆] This research was partially supported by the DFG project ForMooS (grant Ol/98-3).

© Springer-Verlag Berlin Heidelberg 2004

[19,32]). Properties of CSP-OZ can be formally verified, for instance by applying the FDR model checker to the process semantics of CSP-OZ [14,39].

Viewed from the formal method's side, the advantage of a combination with the UML is the possibility of *graphically* specifying the object-oriented and behavioural features of a system, without having to use the less intuitive notations offered by the formal method. The formal specification is then (partially) obtained from the UML diagrams by means of a translation. This may help in gaining acceptance of the use of a formal specification language.

For the software engineering side, the purpose of integrating a formal method into the development process is twofold: on the one hand it is used for supplying UML diagrams with a precise meaning (thus opening the possibility for *verifying* a design); on the other hand it serves as a bridge between the high-level graphical UML model and the final implementation. To preserve the precision of the formal specification in the implementation we take a pragmatic approach: Java programs are annotated with correctness *assertions* using the Java Modeling Language (JML) [24] and CSP$_{jassda}$ [27]. JML offers static assertions like pre- and postconditions and invariants for methods as well as model variables for data abstraction. CSP$_{jassda}$ complements this by offering trace assertions in a CSP-like notation for specifying the required order of method calls. Both kinds of assertions are generated from the formal specification to ensure a tight correspondence. We stipulate that the final Java implementation is hand-written but checked at runtime against these assertions. Thus our approach involves the three levels shown in Fig. 1 (plus the level of Java programs not discussed here).[1]

This approach to system development is tailored to *reactive systems*, the application domain of CSP-OZ. We have therefore chosen a specific interpretation for UML, which fits this domain best. The subset of UML used so far includes class diagrams, state machines, and the structure diagrams of UML-RT [36]. To support the combination with CSP-OZ we are developing a UML profile (with tool support), which provides specific stereotypes and tags for CSP-OZ classes and their ingredients as well as the capsules, protocols and ports of UML-RT structure diagrams. During specification generation every UML class is translated to a CSP-OZ class in which the attribute and method names are extracted from the class diagram, and the CSP part is obtained by a translation of the associated state machine. The structure diagrams describe the architecture of the system; they are translated into a CSP system description involving parallel composition [13]. In the next step, this CSP-OZ specification is used to generate JML and CSP$_{jassda}$ specifications. The final hand-written Java program is checked against the assertions using a runtime-checker for JML [25] and the tool jassda [4,5].

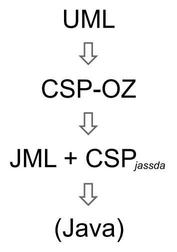


Fig. 1. Development levels

[1] There are parallels to Model Driver Architecture (MDA): Our approach involves platform independent models (PIMs) and platform specific models (PSMs) as well as transformations between them. The UML level is the main level for development.

In this paper we illustrate this approach with the case study "Holonic manufacturing system". The case study originates from the area of production automatisation. In a holonic manufacturing system (HMS) autonomous transportation agents are responsible for the flow of material in a plant. Coordination of transportation and processing is distributed among stores, machine tools and agents without central control. The case study thus falls into the category of reactive systems involving a high degree of parallelism and communication.

The paper is structured as follows: according to the three levels of development the next three sections use parts of the UML model, the CSP-OZ specification and the JML/CSP$_{jassda}$ assertions taken from the full case study to explain the different levels and transformations. It concludes with a discussion of some related work.

2 Modelling

In this section parts of the UML model of the case study are presented. A special profile, which is described next, is used to integrate CSP-OZ with the UML. After the profile the actual model parts are presented, followed by comments on the generation of CSP-OZ definitions leading to the CSP-OZ specification in Sec. 3.

2.1 A UML Profile for CSP-OZ

The UML[2] contains an extension mechanism in form of so-called *profiles*. They consist of *stereotype* and *tag* definitions, and *constraints* concerning the newly introduced items. It is a conservative extension mechanism, in the sense that it only allows for customisation and extension of existing model elements for specific purposes, but must not conflict with standard UML semantics. The stereotypes introduce 'new' model elements, with additional features represented by tags, and constraints to specialise the semantics.

The profile presented here draws on the ideas originally presented with the ROOM method [35] and modified for use with the UML under the name UML-RT in [16,36], especially those ideas dealing with reactiveness, concurrency and distribution. The purpose of the profile is to provide model elements which are suited to modelling of reactive systems *and* close enough to CSP-OZ, so that an automatic generation of CSP-OZ specifications from the model is possible – provided that the modeller does not use extra-profile elements.

The main elements are *capsules*, *ports* and *protocols*. A capsule ('actor' in ROOM) represents a self contained, active unit of computation. It has attributes and methods almost like a typical class of an OO-language, but they are *never* accessible from the outside. Instead of method invocation used with classes, capsules are used with message passing. Message directions and types are specified in protocols; these protocols are referenced from the capsules by their ports. Compatible ports of capsules can be connected to enable communication between them. The architecture of a concrete system is specified using *structure*

[2] In this paper we refer to the current official version 1.5 [38].

diagrams, showing instances of capsules and the connections between their ports. A special type of visualisation is used for these diagrams (see Fig. 3), drawing the ports as small boxes on the border of a capsule and lines between ports to denote communication paths.

In contrast to ROOM/UML-RT we do not use a state machine to control the operation of a capsule by receiving messages and performing actions (internal methods calls, direct modification of the state space, sending messages), but have a strict binding of methods to ports/protocols. We use a state machine for a capsule only as a *protocol* state machine (see Fig. 4), specifying the allowed sequence of communications (with a blocking semantics) without accessing the state space.[3]

All subsequent stereotype and tag definitions are contained in a stereotyped UML package named «profile» CSP-OZ, i.e., a profile definition. Here only the main items are presented in detail. Examples are shown in Fig. 2.

Capsules. A capsule is the main building block of distributed systems. It encapsulates the state and the only way to interact with a capsule is by communication over one of its ports. Capsules can be nested to build complex capsules from simple ones, but this is not visible to the outside; likewise, the surrounding capsule has no special access to the contained capsules – all interaction has to be done via ports.

In the profile the «capsule» stereotype is defined for model elements of type Class. The following constraints apply: A capsule has neither public attributes nor public methods. Only the ports, i.e., the associations with a protocol, stereotyped as «base» or «conjugated», are visible outside. Methods of a capsule are either stereotyped as «base» or «conjugated» matching the stereotype of the port which references a protocol containing the signature for the method. A capsule may only inherit from capsules. This stereotype has the following tags: invariant holds the Z predicate for the class invariant; init contains a Z predicate describing the initial state.

Protocols. A protocol is used to define a binary communication pattern. It distinguishes two sides of the communication, namely, *base* and *conjugated*. The communication pattern is specified from the view of the base role. From the CSP-OZ point of view, a protocol describes a number of channels (one for each operation defined by the protocol) by specifying their channel types. Protocols are primarily used to aid graphical modelling on the UML level and are only represented by definitions of communication channels for their operations in the CSP-OZ classes.

In the profile the «protocol» stereotype is defined for model elements of type Class. The following constraints apply: A protocol has no attributes. It is an abstract class that has no methods, but only operations, which are stereotyped either as «in» or «out». A protocol may only inherit from protocols.

[3] Protocol state machines (enhanced with a blocking semantics) fit very well to the CSP process expressions of CSP-OZ classes.

Ports. Ports are represented as references to protocols. A port is just a concept of modelling complex (typed) communication endpoints and does not appear as an attribute of a CSP-OZ class. Two ports can be connected (on the structure level specifying a concrete instance of a system) only, if they both reference the same protocol *and* one is a base port and the other is a conjugated port. For a conjugated port «in» and «out» decorations of operations are reversed and input parameters become output parameters and vice versa.

In the profile the «base» / «conjugated» stereotype is defined for model elements of type Association. The following constraints apply: A port may only reference a protocol from a capsule, that is, it has exactly two association ends, one connected to a capsule and the other to a protocol. The association end connected to the capsule has aggregation type aggregate and the association is only navigable from the capsule to the protocol.

Communication Directions. In a protocol the «in» / «out» stereotypes described below are used to specify the 'direction' of operations. Although these stereotypes are present in the UML-RT draft(s), the semantics here is slightly different: we do not restrict the communication to simple, unidirectional signals, but preserve method call like communication of CSP-OZ with input and output parameters both possibly occurring in one method call (communication). So in our case these stereotypes specify, whether a communication is passive («in», waiting for a call) or active («out», initiating communication from a state machine) seen from the «base» side of a protocol.

In the profile the «in» / «out» stereotype is defined for model elements of type Operation. The following constraints apply: These operations must be abstract, that is, they have no associated implementation. This stereotype has the following tags: inDecl contains the declarations for the input parameters; outDecl contains the declarations for the output parameters; addr holds the declaration of the parameter used for addressing.

Communication Behaviour. For each operation in a protocol referenced by a capsule via «base» or «conjugated» ports, an equally stereotyped method has to exist in the capsule. It specifies the communication behaviour, i.e., the precondition for the communication to be allowed and the effect on the containing capsule's state space.

In the profile the «base» / «conjugated» stereotype is defined for model elements of type Method. The following constraints apply: These methods must be owned by a capsule and may not be public. This stereotype has the following tags: changes lists the attributes of the capsule owning this method, which might be changed by the method; enable contains a Z predicate specifying the enable conditions for the method; effect contains a Z predicate describing the effect on the state space of the capsule owning this method.

Additional conventions and comments

In the following, additional features of the CSP-OZ adaption which are not part of the profile are discussed. First, in the statechart diagrams for the capsules the following conventions are used:

– Blocking semantics and synchronous communication. Communication between capsules (and therefore state machines) is synchronous and no events are silently discarded.
– Only the names of methods in the corresponding capsule may be used as triggers for transitions. Parameters are always omitted and there are no guards or actions.

The decision against guards and actions was made because it is closer to the CSP-OZ idea of separating data and control. It is of course still an option to include the contents of enable and effect tags of («base», «conjugated») methods as guards and actions in the state machine diagrams, but this would probably make the statecharts less readable.

2.2 UML Modelling for the Case Study

The case study "Holonic manufacturing system" is part of the German DFG priority program "Integration of specification techniques with applications in engineering"[4]. The system consists of 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) and machine tools for processing the workpieces. Every workpiece has to be processed by all the machine tools in a fixed order. 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 machines are empty. When the in-store or a machine contains a (processed) workpiece it broadcasts to all hts a request to deliver this workpiece. The hts (when listening) send some offer to the machines, telling them their cost for satisfying the request. Upon receipt of such offers the machine decides for the best offer and gives this hts the order, which executes it and transports the workpiece to the next machine tool in the processing order. In this way, all workpieces are processed by all tools and transported from the in- to the out-store.

In this section the UML diagrams describing the transport element ('Hts') are presented. For a better readability, these diagrams do not show the internals (tags); for those details refer to section 3.[5]

The class diagram for the transport system is shown in Fig. 2. It contains all capsules used by the transport system and all protocols used by these capsules. Hts is a compound capsule containing instances of three other capsules (HtsCtrl, Driver, Acquisition). It has no attributes or methods of its own and even the ports are derived from the enclosed capsules, that is, messages are simply forwarded from and to the ports of the contained capsules.[6] The internal architecture of Hts has to be specified in a structure diagram (Fig. 3). The three protocols

[4] http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/index.html
[5] UML diagrams typically do not show larger textual items of a model (documentation, method bodies, etc.), since it would make the diagrams unreadable.
[6] The outer ports are therefore called *relay* ports in ROOM and UML-RT.

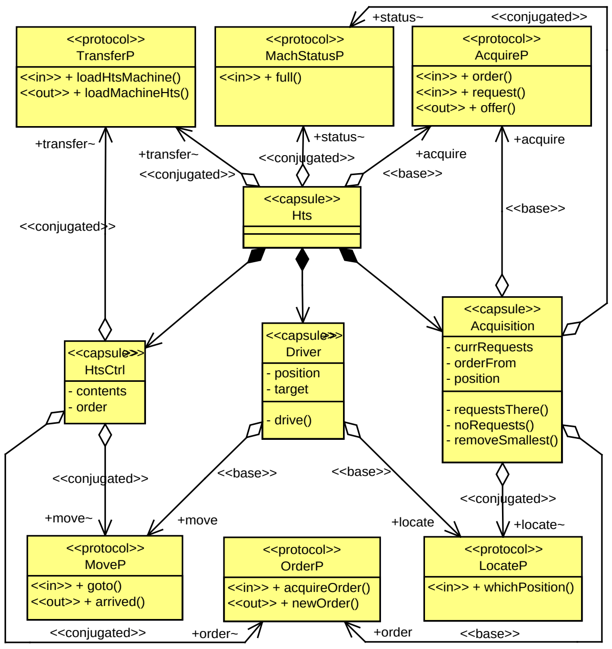


Fig. 2. Class Diagram: Transport

in the upper half of Fig. 2 are the main protocols of the whole system; they are used to communicate with the stores and machines and would show up in the class diagram(s) for the rest of the system as well. On the bottom of the diagram another three protocols are defined; they are used only for (Hts-) internal communication between the contained capsules. Thus Hts has no ports referencing these protocols.

The ROOM/UML-RT structure diagrams (e.g., Fig. 3) are used to specify the (communication) structure of instantiated systems or capsules. Capsules are rendered as boxes, with ports appearing as little boxes on their border. A port's colour visualises whether it refers to the «base» (black) or «conjugated» (white) protocol.

The state machines for the capsules specify the allowed communication sequences. For non-compound, concrete capsules a state machine is mandatory. Fig. 4 shows the state machine for HtsCtrl.

2.3 Translation to CSP-OZ

The translation to CSP-OZ encompasses three tasks: translating the state machines to CSP process expressions, generating CSP-OZ classes from the class diagrams (using the process expressions just mentioned), and translation of the structure diagrams.

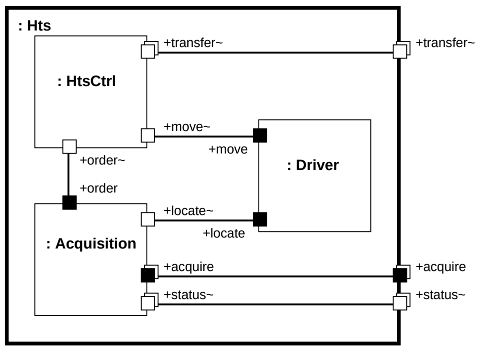


Fig.3. Structure Diagram: Hts

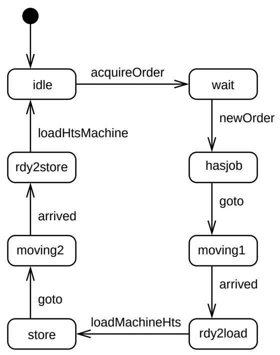


Fig.4. Statechart: HtsCtrl

1. For simple protocol state machines (without concurrency and history, and using only completion transitions on compound states) a straightforward translation to CSP exists: For each state a process is defined as an external choice over all events occurring on transitions originating at the state, prefixed to the processes (represented by their name) for the target state. For the initial state on the top level of the statechart, the process `main` is created; final states correspond to the special CSP process `SKIP`, which represents termination. This can be extended to state machines with a restricted kind of concurrency (disjoint event sets in the concurrent submachines); the processes for the concurrent submachines are then put in parallel.

   The general case needs a non-trivial translation; we have developed such a translation for a larger class of state machines [29]. Since the resulting CSP processes of this general translation scheme are much larger even for the simple state machines above, the simple translation is preferred.

2. Generation of the CSP-OZ classes consists mainly of assembling the tags attached to the various stereotyped model elements. Each operation of a protocol introduces a channel declaration in the CSP-OZ class using the protocol. There are three kinds of channels: `method` channels, which correspond to passive behaviour, i.e., communication will be initiated from the other end; `chan` channels, which correspond to invocation of a (remote) method; and `local_chan` channels, which are used for intra class communication. An operation marked as «in» in a protocol introduces a `method` channel declaration in the CSP-OZ class for a capsule with a «base» reference to this protocol and a corresponding `channel` channel declaration for a capsule with a «conjugated» reference (with the reversal of input and output parameters). With roles reversed, the same happens for operations marked «out». CSP-OZ `inherit` clauses are generated from the Generalization relationships. The CSP part already exists; composition of capsules is handled in step 3.

3. Translation of the structure diagrams is more complex; a general scheme is described in [13].

3 Specification

The CSP-OZ classes of the case study can thus be systematically generated from the UML model. Currently this has to be done by hand but an implementation of the profile within Rational Rose is under development. Next, we illustrate the the translation by looking at the classes *HtsCtrl* and *Hts*.

The translation first generates a given type *CRef* for every defined class C, containing *reference names* of instances, e.g., we have a type *HtsRef* for references to instances of *Hts* and types *MachineRef* and *ActiveMachineRef* for classes *Machine* and *ActiveMachine* which are in the full case study.

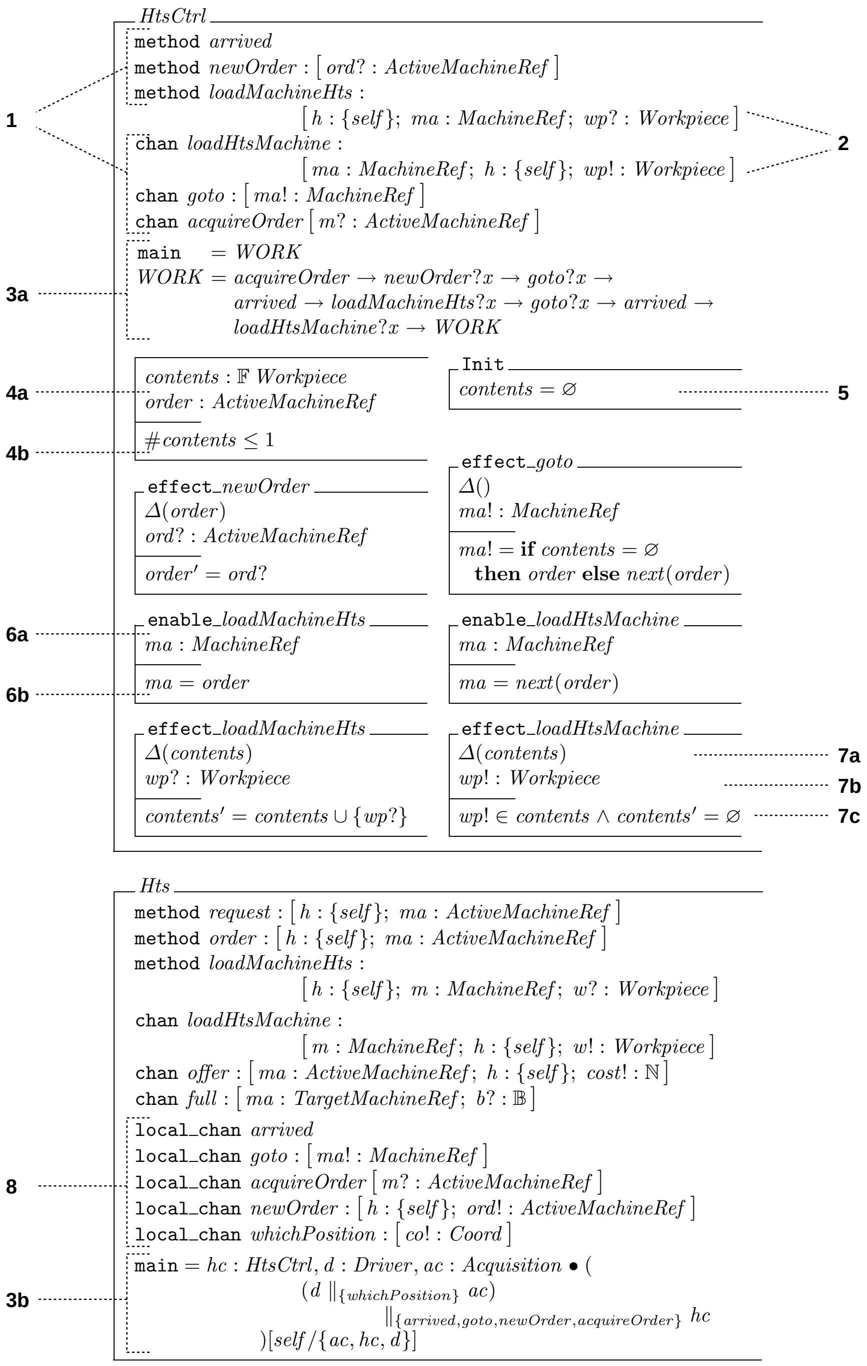
Next, for the capsules *HtsCtrl* and *Hts* in the UML model a CSP-OZ class is generated (Fig. 5). The first part of the specification of class *HtsCtrl* describes the basic interface according to the class diagram (Fig. 2). Then the attributes of the class are defined and the initial state values are given. The last part of class *HtsCtrl* specifies the enabling conditions (guards) and effects for the execution of methods. The variable *self* in the class specification for *HtsCtrl* (Fig. 5) is a special variable holding the instance name of objects of this class; for any instance of this class it can be regarded as a (unique) constant. Here, it is used for addressing objects: when a method is called on a channel and addressing is required (i.e, several objects can be reached via this channel), the first parameter will always be the instance name of the object.[7] This is achieved by restricting the value of the first parameter to *self* in the capsule implementing the '`method`' side of the operation. Another application of *self*, also used in the specification of *HtsCtrl*, is as a caller identification, enabling the callee to refer to the caller later.

To specify the class *Hts* we have to deal with instantiation of the three components *HtsCtrl*, *Driver* and *Acquisition* as depicted in Fig. 3. Since class *Hts* is a "pure" composition of *active* classes, it only has a process specification, but no data part. The interface is the union of the component interfaces, with all channels made local, which are not connected to the (relay) ports of *Hts*. The instantiation of the components is defined local to the main process. Thus the names of the component instances remain local to objects of class *Hts*. The final renaming $[self/\{ac, hc, d\}]$ assures that all references to the instance names are replaced by a reference to the *Hts* object, effectively connecting the relay ports of *Hts* with the ports of the contained capsules.

If all tags in the profile have been properly filled in the UML model the CSP-OZ class specifications in Fig. 5 can be automatically generated. Using the numbers in Fig. 5 we look at the generation of the two classes in more detail:

1. Depending on «in» and «out» stereotypes in the protocol and on the «base» and «conjugated» stereotypes on the ports, a channel is declared as either `method` or `chan`.

[7] The declaration of this parameter is stored in the addr tag for operations on the UML level.


1
2
3a
4a
4b
5
6a
6b
7a
7b
7c
8
3b

```
HtsCtrl
  method arrived
  method newOrder : [ ord? : ActiveMachineRef ]
  method loadMachineHts :
                [ h : {self}; ma : MachineRef; wp? : Workpiece ]
  chan loadHtsMachine :
                [ ma : MachineRef; h : {self}; wp! : Workpiece ]
  chan goto : [ ma! : MachineRef ]
  chan acquireOrder [ m? : ActiveMachineRef ]

  main  = WORK
  WORK = acquireOrder → newOrder?x → goto?x →
           arrived → loadMachineHts?x → goto?x → arrived →
           loadHtsMachine?x → WORK

  | contents : 𝔽 Workpiece
  | order : ActiveMachineRef
  |--------
  | #contents ≤ 1

  Init
  | contents = ∅

  effect_newOrder
  | Δ(order)
  | ord? : ActiveMachineRef
  |--------
  | order′ = ord?

  effect_goto
  | Δ()
  | ma! : MachineRef
  |--------
  | ma! = if contents = ∅
  |   then order else next(order)

  enable_loadMachineHts
  | ma : MachineRef
  |--------
  | ma = order

  enable_loadHtsMachine
  | ma : MachineRef
  |--------
  | ma = next(order)

  effect_loadMachineHts
  | Δ(contents)
  | wp? : Workpiece
  |--------
  | contents′ = contents ∪ {wp?}

  effect_loadHtsMachine
  | Δ(contents)
  | wp! : Workpiece
  |--------
  | wp! ∈ contents ∧ contents′ = ∅
```

```
Hts
  method request : [ h : {self}; ma : ActiveMachineRef ]
  method order : [ h : {self}; ma : ActiveMachineRef ]
  method loadMachineHts :
                [ h : {self}; m : MachineRef; w? : Workpiece ]
  chan loadHtsMachine :
                [ m : MachineRef; h : {self}; w! : Workpiece ]
  chan offer : [ ma : ActiveMachineRef; h : {self}; cost! : ℕ ]
  chan full : [ ma : TargetMachineRef; b? : 𝔹 ]
  local_chan arrived
  local_chan goto : [ ma! : MachineRef ]
  local_chan acquireOrder [ m? : ActiveMachineRef ]
  local_chan newOrder : [ h : {self}; ord! : ActiveMachineRef ]
  local_chan whichPosition : [ co! : Coord ]
  main = hc : HtsCtrl, d : Driver, ac : Acquisition • (
                (d ||_{whichPosition} ac)
                          ||_{arrived,goto,newOrder,acquireOrder} hc
            )[self/{ac, hc, d}]
```

Fig. 5. CSP-OZ specifications for capsules *Hts* and *HtsCtrl*

2. The operation signature (type of data on the channel) is generated using the contents of the inDecl, outDecl and addr tags belonging to the corresponding protocol operation.
3. For a simple capsule, the CSP part of the class contains the translation of the corresponding state machine (**a**); for compound capsules it contains the instantiation of the contained capsules (**b**).
4. The state schema is populated with the attributes of the capsule (**a**) and the class invariant is taken verbatim from the invariant tag of the «capsule» (**b**).
5. The init schema uses the predicate from the init tag of the capsule.
6. Enable schemas are generated using information from the corresponding protocol for the parameter declarations (**a**) and from the method's enable tag for the body (**b**).
7. Effect schemas are generated using the changes tag of the method for the Δ-list (**a**), information from the protocol for the parameter declarations (**b**) and from the method's effect tag for the body (**c**).
8. For internal connections in the structure diagram of a compound capsule local channel declarations are generated.

A larger part of the CSP-OZ specification of this case study can be found in [39]. Using a verification technique for CSP-OZ proposed in [14] the specification of the holonic manufacturing system has been verified, showing for instance deadlock freedom and adherence to the correct processing order.

4 Implementation

To link the CSP-OZ specification with a final Java implementation we now take a different approach. Instead of generating Java implementations from CSP-OZ we generate Java interfaces with *assertions*. The final (handwritten) implementation of these interfaces can then be monitored against the assertions. For writing the assertions we use two intermediate languages that support monitoring viz. runtime checking – JML and CSP$_{jassda}$.

The state-based part of CSP-OZ, formalised in Object-Z, is mapped to assertions written in the Java Modeling Language (JML) [24]. JML is a behavioural interface specification language (BISL), i.e., it provides "rich interfaces" that enrich the syntactic interface of a software module (signature) by a specification of its behaviour in the form of assertions (pre- and postconditions and invariants). These assertions can be checked at runtime [25]: whenever a violation of an assertion is detected an exception is thrown and the program terminates.

However, in JML the order of methods calls cannot be specified directly. To overcome this shortcoming, we complement JML by CSP$_{jassda}$, a generalisation of the trace assertion facility of Jass [2]. *Jassda* stands for *Java with assertions debugger architecture.* It uses the Java Debug Interface (JDI) to enable runtime checking by monitoring a program during execution and comparing the monitored behaviour with the CSP-like specification [5]. The CSP part of a CSP-OZ specification can thus be translated into trace assertions written in CSP$_{jassda}$.

By combining the two runtime checking methods we guarantee that the current program run performs correct data modification (JML) in the correct order (CSP$_{jassda}$) – until we detect a violation of the specification(s).

Both formalisms have in common that the specification of the Java program is separated from its implementation. This allows one to switch to an alternative implementation while keeping the same specification.

4.1 From Object-Z to JML

JML annotates normal Java programs with special types of comments so that the same annotated programs may be used both for compilation by an ordinary Java compiler and for the JML tools. Besides method pre- and postconditions and class invariants, JML also provides *model variables.* These variables are accessible in the specification only and describe the *abstract* state of instances of a type. In contrast to normal Java class attributes these model variables can also be used in interfaces and thus describe (part of) the abstract state of an implementing Java class.

This kind of abstraction, hiding the concrete implementation of the state space, fits very well with Object-Z. It enables us to use an almost direct mapping from the abstract Object-Z state space to the abstract state space of a JML interface, that is finally mapped to the concrete state space of the implementation by an abstraction relation, i.e., a JML *represents clause.*

The translation of CSP-OZ to JML defines a Java interface with JML specification for every CSP-OZ class. These interfaces have to be implemented by the final Java program.

We do not treat the types of reference names in the translation directly, but instead we identify them with the Java types of the classes they are referring to. Whenever we define a Java type `class C` or `interface C`, we define a reference to that type with name `ref` by the declaration `C ref;`.

In the following we use the CSP-OZ class *HtsCtrl* of Fig. 5 as the running example to explain the translation to a JML/Java interface `HtsCtrlSpec`. The interface definition of the CSP-OZ class is translated into the interface of a Java class. Since we want to be able to associate a specification with all possible communications we have to translate both types, `method` and `chan`, to Java methods. The schema type of a channel is translated into the formal parameters and the return value of the method. References to *self* are omitted because a Java class "knows" itself (`this`), other reference parameters become formal parameters of the method. The same holds for the input parameter. The output parameters are translated into the return values of the Java methods. Since Java methods may only have exactly one return value, we have to translate more than one output parameter to a Cartesian product of participating outputs. This is summarised by the following rule.

Translation Rule 1 *A CSP-OZ class or Object-Z class C is translated to a JML interface* `CSpec` *extending either* `JMLType` *or the specification classes from the* `inherit`*-clause. Every method or channel m of C is represented by a Java* *method. Let* $[ref : \{self\};\ other : OtherRef;\ in? : In;\ out? : Out]$ *be the schema type of* m *with* `InType`, `OutType` *as JML types for In and Out, then the Java method will get the signature* `OutType m (OtherSpec other, InType in)`.

Translation of the schemas starts with the state schema that will be transformed into model variables of our specification. In our running example we use JML's `JMLValueSet` for sets of arbitrary objects to model the given type *Workpiece*. Since Java does not provide generic types, like C++ templates, we cannot reflect the type of *contents*' elements more precisely[8]. The predicate part of the schema and the implicit predicates that we get by normalisation of the schema are translated into class invariants. Finally, the init schema is translated into initial conditions for the model variables.

Translation Rule 2 *The declarations within the state schema of a CSP-OZ class or Object-Z class become model variables in the JML specification of the class. The predicates of the normalised state schema become class invariants of the JML type. The predicates of the normalised* ***Init*** *schema are translated into* initially clauses *of the model variables.*

To complete the "rich" interface `HtsCtrlSpec` we add the method specifications. For each operation the enable schema yields the precondition of the method indicated by the keyword `requires`. The effect schema is translated into the postcondition indicated by the keyword `ensures`, where all references to the pre-state (the unprimed variables) have to be enclosed by the operator `\old()`.

Translation Rule 3 *For every channel or method m we translate the* ***enable_m*** *and* ***effect_m*** *schemas to precondition, postcondition and assignable clause of the Java method* `m`*. For the precondition we have to translate* ***enable_m*** *to a Java boolean expression. For the postcondition we translate* ***effect_m*** *to a Java boolean expression where references to the pre-state are enclosed by* `\old()`*. The assignable clause is just the* Δ*-list.*

Fig. 6 shows (part of) the JML specification that the translation rules produce for the CSP-OZ class *HtsCtrl*. In some cases we cannot translate the specification directly. In particular, set comprehensions pose a problem because they imply a quantification over the set's elements. In that case we have to find an equivalent specification that makes the quantification explicit and thus can be translated to JML expressions. In the case study there was only one problematic expression for which it was simple to find an equivalent translatable expression.

4.2 Linking JML specification and implementation

To link an actual implementation with our JML specification two things need to be done. We must implement the interface(s) of our JML specification and

[8] A generic type concept `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html`) will be part of the upcoming J2SE 1.5 Release. Also some investigations of the formal aspects of such a facility were done.

```
package hms.spec;
//@ model import org.jmlspecs.models.JMLValueSet;
//@ model import org.jmlspecs.models.JMLType;

public interface HtsCtrlSpec /*@ extends JMLType @*/{

        //@ public model instance JMLValueSet contents
        //@ public initially contents.isEmpty();
        //@ public model instance MachineSpec order;

        //@ public invariant contents.size() <= 1;
        ...
        /*@ public normal_behavior
              requires   ma.equals(order);
              assignable contents;
              ensures    contents.equals(\old(contents.union(new JMLValueSet(wp)))); @*/
        public void loadMachineHts(MachineSpec ma, WorkpieceSpec wp);

        /*@ public normal_behavior
              requires   ma.equals(order.next());
              assignable contents;
              ensures    \old(contents).has(\result) && contents.isEmpty(); @*/
        public WorkpieceSpec loadHtsMachine(MachineSpec ma);
        ...
}
```

Fig. 6. JML specification `HtsCtrlSpec` for class *HtsCtrl*

thus provide an implementation for the methods specified there. Second, we have to fill the specification with life by giving the relation between model variables occurring in the JML specification and attributes of the implementing classes. The implementation should be a *data refinement* of the JML specification. At runtime it is checked that the concrete implementation adheres to the abstract specification via a representation relation. The JML `represents` clause links model variables with the implementation and thus defines this representation relation.

This clause usually is a function mapping attributes of the implementing class to the model variables by giving an expression that returns the required type. It is also possible to use a representation *relation*, for which JML provides the keyword `\such_that`. Proper representation relations are, however, hard to handle in automatic program verification as shown in [3] and for the same reasons in a runtime assertion checker and thus should be avoided.

Below, the representation relation between a class *HtsCtrl* and the JML specification `HtsCtrlSpec` is given. The class defines a concrete attribute `cont` that is used to represent the model field `contents`. The expression `(cont == null ? new JMLValueSet() : new JMLValueSet(cont))` maps the concrete attribute to an appropriate type for the model field.

```
package hms.impl;
import hms.spec.*;
//@ model import org.jmlspecs.models.JMLValueSet;

public class HtsCtrl extends Thread implements HtsCtrlSpec {
        protected WorkpieceSpec cont;
        /*@ protected instance represents contents <-
                (cont == null ? new JMLValueSet() : new JMLValueSet(cont)); @*/
```

4.3 From CSP to CSP$_{jassda}$

In this section we will show how to link the specification with the dynamic behaviour of the implementation. In CSP-OZ we use CSP processes to specify the order of communications between active objects, thus giving an event based view on the specification.

In the JML part we mapped communication channels to methods of Java interfaces. To observe communication of our objects we have to observe invocations of these methods. More precisely, the events to observe are start and end of a method invocation (a method is pushed to the method stack or popped). A communication from a sender to a receiver will result in four events: the communication starts at the caller's side before it starts at the receiver's side, and both calls end before a new one begins.

To specify such an order on method start and end events we developed CSP$_{jassda}$ [27]. This language is, like JML, somewhat specific to Java[9] and very close to CSP so that it will be easy to translate the specification. The design of CSP$_{jassda}$ is closely coupled with the development of the tool `jassda` [4,5]. The `jassda` tool is able to test whether a program generates a given set of events in the correct order according to the specification given in CSP$_{jassda}$. This test is performed at the byte-code level so that even for a different set of events no modification or instrumentation of the code is required.

In every class of the specification the CSP process `main` describes the possible communication behaviour of that class and thus the order of method invocations at the JML/Java level. For the translation we have to distinguish two types of processes that we get from our translation: *simple class processes* and *instantiation processes*. A simple class process describes the behaviour of one instance of a class without any assumption about its environment. Instantiation processes occur for every instantiation of subcomponents of a class, i.e., they result from the translation of structure diagrams.

Simple class processes. A simple class process just describes the order of events that are produced by exactly one instance of exactly that class. Thus we need to allow this main process to run for every instance. This is done by a parallel composition of a parameterised process that is instantiated for each instance of a class that emits a relevant event. The parameter is used to restrict the alphabet of that subprocess to the events of one instance of the class.

In a simple class process only methods of the class are mentioned and calls to these methods must end before we call another method. Therefore we abbreviate the CSP$_{jassda}$ specification with one that has the same structure as the CSP-OZ process. To use the `jassda` trace checker we expand the abbreviated specification to the full CSP$_{jassda}$ specification by a preprocessor. For example, the simple class process of *HtsCtrl* (cf. Fig. 5) is translated into the following abbreviated CSP$_{jassda}$ specification.

[9] Although it will be possible to transfer most of the concepts to another programming language.

```
public interface HtsCtrlSpec {
    public void acquireOrder();
    public void newOrder(ActiveMachineSpec newOrder);
    public MachineSpec gotoMachine();
    public void arrived();
    public void loadMachineHts(MachineSpec ma, WorkpieceSpec wp);
    public WorkpieceSpec loadHtsMachine(MachineSpec ma);
    /*% public csp
        main() { WORK() }
        WORK() { acquireOrder -> newOrder -> gotoMachine ->
                 arrived -> loadMachineHts -> gotoMachine -> arrived ->
                 loadHtsMachine -> WORK() }   %*/
}
```

Instantiation processes – synchronisation and delegation. To translate a class that instantiates other classes, i.e., processes, we have to take the interfaces of the instantiated processes into account. In our example *Hts* instantiates *HtsCtrl* with reference name *hc*, and *Driver* with reference name *d* and *Acquisition* with reference name *ac* (cf. Fig. 5). The connection diagram of this CSP-OZ process corresponds to the structure diagram of class *Hts* (Fig. 3). The Java methods represent the ports from the diagram. A communication event on a channel is represented by four events of the corresponding Java methods (start and end of the methods on both sides of the channel). Thus a CSP channel is specified by an order on these events. So we specify a recursive processes for each channel and compose them in parallel.
For every such instantiation process we have three types of channels:

– For a channel that is local to one of the instances there is no connection to model and thus it is not considered.
– For every channel m that instances a of a CSP-OZ class A and b of a CSP-OZ class B synchronise on we need to link the Java methods `m` of classes `ASpec` and `BSpec`.
  The synchronisation is represented by subsequent calls to the methods of the two classes where we pass the result of the first call of the method (the output of the `method` side) as argument to the call of the second method (the input of the `chan` side). The process accepting this kind of event traces is abbreviated to `SYNC(a,b,m)`.
  In our example the instances *hc* (class *HtsCtrl*) and *d* (class *Driver*) synchronise on channel *arrived*, so we add the process `SYNC(hc,d,arrived)` to the specification of *Hts*.
– If CSP-OZ class C instantiates a class A with reference name a and a channel m of A is visible to the outside world it has to be "exported" by the Java class `CSpec`. In this case `CSpec` must forward the call to the instance a (that provides the method). Forwarding means that `CSpec` provides a method `m` with the same signature as method `m` of `ASpec`, every call to `CSpec.m` is followed by a call to `a.m` and the latter call ends before the first one (abbreviation: `EXPORT(a,m)`).
  For the class *Hts* of the case study this means that channel *offer* has to be exported from instance *ac* and therefore we add an `EXPORT(ac,offer)` process to the specification of *Hts*.

Using this translation we obtain a CSP$_{jassda}$ process for every class describing the communication behaviour of every instance of this class. To obtain the specification of the whole system we let all these processes run in parallel.

4.4 Execution with Runtime Checks

Fig. 7 summarises the process from CSP-OZ to Java executables with runtime checks using JML and jassda. We translate the CSP-OZ specification into a CSP$_{jassda}$ and a JML specification, i.e., Java interfaces annotated with assertions. Then we have to provide Java code that implements[10] the (Java part of the) JML specification. We translate both to Java byte code using the JML runtime assertion compiler and get a byte code program that performs checks for the data part at runtime. Then we run the program while it is connected to the jassda tool that checks the dynamic behaviour against the CSP$_{jassda}$ specification.

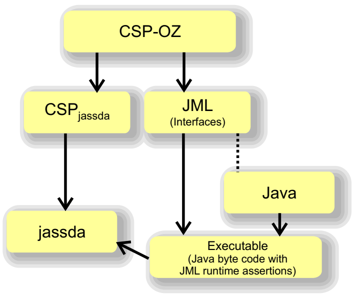


Fig. 7. From CSP-OZ to Java with runtime checks

This approach lets the user run the program as if executed without the runtime checks and thus requires inputs to be given by the user. For reactive systems, like the one from the case study, there is no external input and thus this is no limitation. Utilising the `jml-unit` tool the generated JML specifications would also support unit tests. In that case the expected output would be derived from the specification. Generating appropriate test patterns is still needed in this case. Generating test patterns could benefit from the formal CSP-OZ model, but this is another subject of research.

5 Conclusion

In this paper we described part of the modelling, specification and implementation of the case study "Holonic Manufacturing System". The case study served as an illustration for our approach of integrating a formal method, here CSP-OZ, into a UML and Java based software development. It demonstrated how UML's multi-view modelling facilities (static *and* dynamic behaviour) can be adequately reflected in a formal specification (using an *integrated* formal method) as well as monitored in the final implementation (using one tool for runtime checking of static and another for dynamic behaviour).

Tool support. As a platform for integration of the CSP-OZ profile presented here we have chosen the UML tool *Rational Rose*. Its *extensibility interface* allows the portable customisation of the user interface, implementing new menus and dialogs, and the generation of CSP-OZ code directly from the editor. A first prototype of a CSP-OZ tool in Rose exists, but it does not yet support the full profile. The idea of extending a UML tool with facilities for editing Z or

[10] This means implements in the sense of Java's `implements`.

Object-Z specifications appeared already in [10] but this work covered neither state machines nor structure diagrams. Another work in this direction is [37] which translates UML diagrams (annotated with B) into B.

Related work. Developing formal semantics for UML is currently a very actively pursued research topic (see for instance [23,34,31]). Especially CSP is a prominent choice as a semantic model for UML diagrams [7,11]. Here, we differ in that our choice for CSP-OZ enables a usage of different, orthogonal diagram types of UML: CSP-OZ can be used to give a semantics to class diagrams (modelling static behaviour) in combination with state machines (dynamic behaviour) and structure diagrams (modelling the architecture). The basic ingredients of UML needed for modelling reactive systems are thus covered. The translation to CSP-OZ furthermore opens the possibility of formally checking consistency between the different diagrams [30].

In the context of Java, formal approaches are most often applied in order to verify Java programs. These approaches either use a Hoare-logic for specifying properties and develop proof support with theorem provers [28,20] or they apply model checking techniques to Java programs [17]. An approach combining theorem proving with full automation is the static checker ESC/Java [26].

Besides JML a number of other languages and tools exist which provide "Design-by-contract"-extensions for Java (for instance [22]). JML is, however, the most widely accepted language, and its concept of model variables proves to be an ideal tool for bridging the gap between formal specification and programming language. For runtime verification of the *dynamic* behaviour of Java programs there are tools that monitor programs against temporal logic formulae [8,18]. In the context of UML which specifies dynamic behaviour using state machines (or other interaction diagrams) runtime monitoring against CSP expressions as done by `jassda` seems more adequate.

A different approach for combining CSP-OZ with Java is taken by [6] which transform CSP-OZ specifications into CTJ, an extension of Java with CSP-like processes and channels, using a number of refinement rules.

However, none of the works mentioned above combine all three levels in one approach. Our primary aim is to smoothly integrate the formal method into a software development process with UML *and* Java. Both the modelling and the implementation should benefit from the formal specification, achieving a higher degree of correctness in the resulting design and software.

Acknowledgements: Thanks to John Knudsen for helpful comments.

References

1. E. Ábrahám-Mumm, F.S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *FoSSACS 2002*, volume 2303 of *LNCS*, pages 4–20. Springer, 2002.
2. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In Klaus Havelund and Grigore Roşu, editors, *ENTCS*, volume 55. Elsevier, 2001. `http://www.elsevier.nl/locate/entcs/volume55.html`.

3. C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Workshop on Formal Techniques for Java-like Programs – FTfJP'2003*. ETH Zürich, July 2003. Technical Report 108.
4. M. Brörkens. Trace- und Zeit-Zusicherungen beim Programmieren mit Vertrag. Master's thesis, Univ. of Oldenburg, Dept. of Computing Science, January 2002.
5. M. Brörkens and M. Möller. Dynamic Event Generation for Runtime Checking using the JDI. In Klaus Havelund and Grigore Rosu, editors, *ENTCS*, volume 70. Elsevier, 2002. `http://www.elsevier.nl/locate/entcs/volume70.html`.
6. A. Cavalcanti and A .Sampaio. From CSP-OZ to Java with Processes. In *Workshop on Formal Methods for Parallel Programming, held in conjunction with International Parallel and Distributed Processing Symposium*. IEEE CS Press, 2002. Contained in IPDPS collects proceedings CD-ROM.
7. J. Davies and Ch. Crichton. Concurrency and Refinement in the Unified Modeling Language. In J. Derrick, E. Boiten, J. Woodcock, and J.von Wright, editors, *ENTCS*, volume 70. Elsevier, 2002.
8. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Modelchecking and Software Verification*, number 1885 in LNCS, pages 323–330. Springer, 2000.
9. 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.
10. S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An overview of RoZ - a tool for integrating UML and Z specifications. In *12th Conference on Advanced information Systems Engineering (CAiSE'2000)*, 2000.
11. G. Engels, J. Küster, R. Heckel, and L. Groenewegen. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In *9th ACM SigSoft Symposium on Foundations of Software Engineering*, volume 26 of *ACM Software Engineering Notes*, 2001.
12. 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.
13. C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE'01)*, volume 2029 of *LNCS*, pages 91–108. Springer, 2001.
14. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proc. 1st International Conference on Integrated Formal Methods (IFM)*, pages 315–334. Springer, 1999.
15. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, June 2000.
16. G. Gullekson. Designing for concurrency and distribution with Rational Rose RealTime. Technical report, Rational Software, 2000.
17. J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In K.G. Larsen, editor, *CONCUR 2001*, LNCS. Springer, 2001.
18. K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Rosu, editors, *ENTCS*, volume 55. Elsevier, 2001.
19. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
20. M. Huisman and B. Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 284–303. Springer, 2000.

21. B. Jacobs, J. van den Berg, M Huisman Martijn van Berkum, U. Hensel, and H .Tews. Reasoning about Java classes (preliminary report). In *Proc. OOPSLA 98*, volume 33 of *ACM SIGPLAN Notices*, pages 329–340, Oct. 1998.
22. R. Kramer. iContract - the Java Design by Contract tool. Technical report, Reliable Systems, 1998.
23. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11:430–445, 1999.
24. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for java. Technical Report 98-06v, Iowa State Univ., Dept. of Computer Science, May 2003. See `http://www.jmlspecs.org`.
25. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accomodates both runtime assertion checking and formal verifikation. In *FMCO'02*, LNCS, 2003. To appear.
26. K. R. M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 157–175. Springer, 2001.
27. M. Möller. Specifying and Checking Java using CSP. In *Workshop on Formal Techniques for Java-like Programs – FTfJP'2002*. Computing Science Department, University of Nijmegen, June 2002. Technical Report NIII-R0204.
28. A. Poetzsch-Heffter and J. Meyer. Interactive verification environments for object-oriented languages. *Journal of Universal Computer Science*, 5(3):208–225, 1999.
29. H. Rasch. Translating UML state machines into CSP. Technical report, Universität Oldenburg, 2003.
30. H. Rasch and H. Wehrheim. Checking Consistency in UML Diagrams: Classes and State Machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'03)*, volume 2884 of *LNCS*, pages 229–243. Springer, 2003.
31. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*. Springer, 2000.
32. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
33. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.
34. T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. In S.D. Stoller and W. Visser, editors, *ENTCS*, volume 55. Elsevier, 2001.
35. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
36. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime, 1998.
37. C. Snook and M. Butler. Tool-Supported Use of UML for Constructing B Specifications. Technical report. `http://www.ecs.soton.ac.uk/~mjb/U2Bpaper2.pdf`.
38. OMG Unified Modeling Language specification, version 1.5, March 2003. `http://www.omg.org`.
39. H. Wehrheim. Specification of an automatic manufacturing system – a case study in using integrated formal methods. In T. Maibaum, editor, *Fundamental Approaches of Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 334–348. Springer, 2000.