From CSP-OZ to Java with Processes

Ana Cavalcanti Augusto Sampaio
Centro de Informática/UFPE
P.O. Box 7851 50740-540 Recife PE Brazil
e-mail: {alcc,acas}@cin.ufpe.br

August 25, 2000

Abstract

The language CSP-OZ has been proposed as a combination of the more traditional specification languages CSP and Object-Z, with the objective of facilitating the definition of state and communication aspects of concurrent systems. As a conservative extension of CSP and Z, CSP-OZ is appealing to the users of these languages. In this paper we present rules that support a transformational approach to the development of concurrent Java programs from CSP-OZ specifications. Even though Java is a very successful language, its facilities for communication are very restricted and we make use of a library that supports the use of the concept of processes when writing Java programs. Our work is based on an existing refinement calculus for Z, but includes novel rules to deal with classes and CSP processes.

Keywords: formal methods, program development, refinement laws, object-orientation.

1 Introduction

Following the tendency to join formalisms to obtain languages and techniques that apply to wider classes of applications, several combinations of a process algebra [12, 14, 17] and Z [3] have been proposed. Among them, we distinguish CSP-OZ [9], a language which combines CSP and Object-Z [4]. Its syntax and semantics is very close to that of CSP and (Object-)Z; those acquainted with these languages should have no difficulties to learn CSP-OZ. Moreover, the use of existing tools for CSP to reason about CSP-OZ specifications is a reality [16].

In this paper, we present rules that support a transformational approach to the development of Java [10] programs from CSP-OZ specifications. The strategy is based on an existing refinement calculus for Z [8].

As CSP-OZ is an object-oriented language, it is only natural to think of an object-oriented programming language as a target; the wide acceptance of Java in the programming of distributed systems makes it a good candidate. However, the facilities for concurrency available in Java are based on the concept of monitor and are not in direct correspondence with the idea of process in CSP and CSP-OZ. We chose, therefore, to use CTJ [11], a library that provides a model of processes and channels for Java based on occam [13].

Actually, we extended CTJ to include a more flexible communication mechanism. The communication and process model of occam is very close to that of CSP. As explained later, however, we need a more powerful control over the communications to implement CSP-OZ specifications more directly, since the state part can interfere and possibly block a communication.

The problem of translating CSP based specifications to programs that use occam constructs may seem simple, as the design of occam is based on CSP. Nevertheless, CSP provides many more communication primitives. We show how to implement some of them in occam and discuss the difficulties of implementing others. Moreover, CSP-OZ specifications separate state and communication aspects of a system behaviour. In implementing these specifications in Java, we have to take into account both views of the system.

As usual in modern techniques of program development [15], we use a unified language of specification and programming. It is called COZJAVA and includes the specification structures of CSP-OZ and the programming constructs of Java. In this language, a command can be a Z schema, a CSP process, or a traditional Java command. In a development process, we use a specification written in a pure CSP-OZ subset and finish with an executable program written and Java (with the facilities of the CTJ library).

In the next section we introduce CSP-OZ through a simple example: an instance of the producer/consumer protocol. The CTJ library is briefly described in Section 3. The central contribution of this paper, the strategy for translating CSP-OZ specifications into Java programs, is presented in Section 4. Finally, in Section 5 we summarise the results of our research and discuss topics for further research.

2 CSP-OZ

The design of CSP-OZ [9] aimed at an integrated language intended to be a conservative extension of both CSP and Object-Z. A specification unity is based on the syntax of a class declaration in Object-Z, but extended with a clause for channel declarations and ordinary CSP equations which define the control behaviour of the process. By convention, the keyword `main` is the name of the process which starts the control flow.

An overview of CSP-OZ is given based on the classical producer/consumer example, where the producer stores information into a buffer from which the consumer retrieves. The specification of the *Producer* process in CSP-OZ is as follows.

Producer

$\mathtt{channel}\ put : [a : \mathbb{Z}]$

$\mathtt{main} = put \rightarrow \mathtt{main}$

$val : \mathbb{Z}$

$0 \leq val < 10$

INIT

$val = 0$

$\mathtt{com\_}put$

$\Delta(val)$

$a! : \mathbb{Z}$

$a! = val$

$val' = val + 1$

Each channel has an associated record type: a simplification of schema types in Z where a predicate might be used to constrain the values of the components. The channel *put* above communicates values of a schema type whose singleton component is an integer. The `main` process in this case recursively synchronises with the environment through channel *put*.

The Object-Z part complements the control part with a state space and operations which define the state changes and the communication effects associated with the occurrence of each

CSP event. The first schema above defines the state space of the process, which has a single integer component val; the associated predicate constrains the values that val can assume to the interval 0..10. An alternative would be using this interval in the declaration of v, avoiding the explicit predicate. Nevertheless, as explained later, our translation strategy applies to CSP-OZ specifications that are *concrete* in the sense of using only data types available in Java. Of course, in an initial abstract specification of the system this will not be typically the case; several steps of data refinement might be necessary to transform the initial specification into a *concrete* one.

The next schema initialises the state; this initialisation happens before the occurrence of any event. Each schema that defines the state change and communication effect associated with the occurrence of a CSP event has the name of the event prefixed with the keyword `com_`. The schema `com_`*put* determines that the communicated value through *put* is the current value of *val*, and that this value is incremented every time the event *put* occurs.

The specification of the *Consumer* process is similar, but instead of sending values through the channel *put*, it receives values through the channel *get*. Input varibles can be decored with ? or simple (undecorated) variables, like the variable b in schema *com_get* of the process *Consumer*. There is an important technical difference between decorated and simple input variables. If the restrictions imposed on the simple variables, if any, are not satisfied, the communication is blocked. On the other hand, if a restriction imposed on a decorated input variable is not satisfied, the communication can happen, but as a result the process diverges.

Consumer

$\texttt{channel}\ get : [b : \mathbb{Z}]$

$\texttt{main} = get \rightarrow \texttt{main}$

$val : \mathbb{Z}$

$0 \leq val < 10$

INIT

$val = 0$

`com_`*get*

$\Delta(val)$
$b : \mathbb{Z}$

$val' = b$

In this process, if the value of b is not greater than or equal to 0 and less than 10, the communication through *get* is blocked. If, however, b were a decorated input variable $b?$, the communication would not be blocked, but *Consumer* would diverge after such a communication.

As a matter of fact, in a CSP-OZ process specification, each event c is associated with two schemas: `enable_`c and `effect_`c. The former specifies the situations in which the communication through c is enabled, and the latter specifies the associated effect on the state. By defining a `com_`c schema, we are implicitly defining the `enable_`c and `effect_`c schemas as follows.

$$\texttt{enable\_}c = \exists\, s_i' : Type_{s_i};\ p_j? : Type_j;\ p_k! : Type_k \bullet com\_c$$
$$\texttt{effect\_}c = \texttt{com\_}c$$

The definition of `enable_`c is similar to that of a Z pre-condition [18]. Nonetheless, the decorated input variables are quantified, since restrictions on them are not relevant to determine whether the communication is enabled or not. The definition of `effect_`c is the schema `com_`c itself.

The *Buffer* process receives messages from the producer, through the channel *put*, and sends messages to the consumer, through channel *get*.

Buffer

$\texttt{channel}\ put : [a : \mathbb{Z}]$
$\texttt{channel}\ get : [b : \mathbb{Z}]$

$\texttt{main} = put \rightarrow get \rightarrow \texttt{main}$

$contents : \mathbb{Z}$
$0 \leq contents < 10$

com_put

$\Delta(contents)$ $a : \mathbb{Z}$
$contents' = a$

com_get

$\Xi(contents)$ $b! : \mathbb{Z}$
$b! = contents$

The buffer stores a single message at a time.

The top level specification of our simple system is the parallel composition of the three processes, hiding the *get* and the *put* channels so that they become internal.

$$ProdCons = ((Consumer \parallel Producer) \parallel Buffer) \backslash \{get, put\}$$

We assume that the CSP part of the class specifications is as presented in [12]. The alphabets of the processes are implicitly defined as the sets of events in which they can engage.

As in [12], we assume that each channel is used as either an input or an output channel, but not as both. This is a restriction imposed in [12] and relaxed in [17]. As shown in [17], the theory of processes can be elegantly formulated without restricting the communications. However, if we are interested in an implementation (in occam), we have to assume that the channels are used for communication in only one direction and between only two processes.

3 CTJ

The Communicating Threads for Java (CTJ) package has been designed as an effort to raise the level of abstraction of concurrent programming in Java from threads to processes. Since, conceptually, a process has only one encapsulated thread of control, it gives the developer higher level control over synchronisation, especially when this is purely established by channel communication in the style of CSP. In CTJ, channels are special objects which provide syncronised communication between processes in a predefined way.

As a simple example, consider the process Producer below.

```
class Producer implements csp.lang.Process {

   private ChannelOutput_of_Integer channel;

   public Producer (ChannelOutput_of_Integer chanout) {
      channel = chanout;
   }

   public void run()
   {
        Integer message = New Integer();
        object.value = 100;
        channel.write(message)
   }
}
```

A process must implement the `csp.lang.Process` interface of the CTJ package. Processes communicate via input and/or output channels. These are introduced as attributes which are initialised from the parameters of the class constructor, as illustrated above where the attribute channel is initialised from `chanout`. Following good object-oriented practice, it is recommended that channel attributes be declared as local (private) to the process class.

Every process must implement the method `run()`. The implementation of `run()` captures the behaviour of the process. The communication primitives (`read` and `write`) are methods of the `ChannelInput` and `ChannelOutput` interfaces. There are extensions of these interfaces for particular types of messages, like the `ChannelOutput_of_Integer` used by the `Producer` process.

A network of processes can then be defined by connecting processes developed in the style illustrated by the `Producer` process. Processes can be combined in sequence, in parallel or (nondeterministically) using an alternation construct similar to the one available in occam [13]. The synchronisation between processes is achieved by declaring channels at the network (top) level. For example, to synchronise a producer as defined above with a consumer process, one declares a single channel and uses a same created instance of the channel as argument to the constructors of the producer and consumer processes. Since channels are objects, in this way a reference to the created channel is passed to the producer and the consumer, establishing the desired synchronisation, as shown below.

```
class ProdCons implements csp.lang.Process {
     Channel_of_Integer sync = new Channel_of_Integer()
     Parallel prodcons = new Parallel(new Process[] {
                                new Producer(sync),
                                new Consumer(sync) });

     public void run() { prodcons.run(); }
}
```

The parallel process `prodcons` is an instance of the `Parallel` class whose constructor takes an array of processes as argument. The `prodcons` process becomes active when its method `run()` is invoked. The effect is that the instances of `Producer` and `Consumer` will execute in parallel. The `prodcons` process terminates successfully when all its argument processes do.

4 The Transformation Strategy

The objective of our strategy is to transform a CSP-OZ specification into a Java program that uses the CTJ library. It is based on rules that transform CSP-OZ classes into Java classes that implement the interface `Process`.

Our strategy is based on a structural translation that concentrates on the transformation of classes: it transforms individual CSP-OZ classes to corresponding Java classes. This class-based approach is in tune with the extensibility and reusability aims of object-orientation. The extension of a CSP-OZ specification by the addition of classes does not invalidate previous efforts to obtain a Java program. Furthermore, classes reused across specifications can have their implementations reused as well. We aim at a compositional transformation strategy.

As already mentioned, The starting point of a program development using COZJAVA is a concrete CSP-OZ specification that makes use only of data types available in Java. Typically, such a specification is obtained as the result of applying data refinement techniques of CSP-OZ [9] to a more abstract specification. Our emphasis in this work is not on the well-known techniques of data refinement that are applicable nor on the algorithmic refinement of Z operations, but on structural translation rules.

When a COZJAVA command p_2 implements a COZJAVA command p_1, we write $p_1 \sqsubseteq p_2$. In the next section we present a rule that considers the implementation of a CSP-OZ class by one with the structure as in Java, but whose methods bodies still include schemas and processes. Section 4.2 presents rules that transform CSP-OZ class expressions into Java commands. In the following section, we consider rules to transform CSP processes into CTJ processes. Finally, Section 4.4 introduces rules that apply to schemas. Typically, to tranform a CSP-OZ specification into a Java program we need to apply rules from all these sets.

4.1 Transformation of Classes

First, we consider a CSP-OZ class C defined in a box. It is translated to an homonimous Java class declared as follows.

```
class C implements csp.lang.Process {
```

The Java class, as the CSP-OZ class, is a process.

If C includes an axiomatic description defining a constant c, its value v has to be given explictly in an equality. This is transformed into `final static` $\mathcal{T}$`(`*Type*`) cons = v;`. The function $\mathcal{T}$ translates a CSP-OZ type to a Java type. For instance, it translates $\mathbb{Z}$ to `Integer`. We refrain from using the primitive type `int` because CTJ provides channels that communicate values of the type `Integer`. In our examples, however, for the sake of clarity, we use `Integer` objects as if they were primitive values of type `int`. As we assume that the CSP-OZ specification uses types available in Java, only, the definition of $\mathcal{T}$ is not a problem.

The channels defined in C are translated to private channel components. This possibly requires the definition of subclasses of the CTJ class `Channel`. We assume that C introduces a collection of channels c_i. The values communicated through these channels are bindings of their schema type. Therefore, we need to define a Java class `Channel-of-Type-ci`, which implements channels of the appropriate type. The definition of this class is similar to that of `Channel-of-Integer`, available in CTJ. It uses a class `Type-ci` that implements the schema type of c_i and, for instance, defines its components as public attributes.

This may lead to a program containing many similar classes that implement channels: one for each schema type used in the declaration of a channel in the CSP-OZ specification. This is a weakness of Java, that does not include facilities for defining a generic class `Channel-of-T` implementing channels that communicate values of an arbitrary type T.

Assuming `Channel-of-Type-ci` is available, we can declare c_i as a private attribute of `C`.

```
private Channel-of-Type-ci ci;
```

Local channels of CSP-OZ are translated in the same way, but we assume that all of them have the empty schema as their type: no value is communicated through them and so they are used only as synchronisation points.

To understand the reason for this assumption, suppose that a local channel lc is regarded as an input or output channel. Since it is local, it cannot communicate with processes specified by other classes. Therefore, it could only be used to the internal communication of values. A communication through lc, however, can only occur if a process that uses lc as an input channel is composed in parallel with another process that uses lc as an output channel. Nevertheless, there can only be one `effect_`lc schema, and, in that schema, the use of lc to input or output values is fixed. Therefore, lc cannot communicate values even internally.

Usually, in CSP-OZ, when the type of a channel is the empty schema, its type declaration is omitted. In Java, we declare their types to be `Channel-of-Any`.

The state components s_i of C are declared as private attributes as well. We assume that there are no state components named after channels.

The schema *INIT* of C, that specifies the initial state of its components is translated to a class constructor. Because a constructor in Java cannot return any results, we assume that *INIT* does not have output parameters. The arguments of the corresponding constructor are the input parameters $inp_i? : Type_{inp_i?}$ of *INIT* and the channels.

```
public C(𝒯(Type_inp_i) inpi?; Channel-of-Type-ci ci?)  {
    INIT;
    ci = ci?;
    cj = new Channel-of-Any();  }
```

In the body of the constructor, the schema *INIT* is used as command that initialises the attributes corresponding to the state components. The assignments `ci = ci?` initialise the channel components. Local channels c_j are created by the assignments `cj = new Channel-of-Any()`. The *INIT* schema is later transformed into Java commands using the rules discussed in Section 4.4.

The `run` method of `C` is determined by the CSP part of the specification. It consists of an equation `main` $= P$, which defines a process named `main` using a CSP process expression P, possibly followed by a sequence of equations $NP_i = P_i$ that define processes NP_i as P_i. Each of these processes give rise to private inner class declarations.

Before introducing class declarations, however, we apply transformation rules to the equations to eliminate possible uses of the change of symbol, labelling, and interrupt operators, since these are not available in CTJ. The change of symbol operator amounts only to a renaming of channels, and labelling can be expressed using the change of symbol operator. The rules to eliminate the use of these operatores are simple and standard. For the sake of conciseness, we omit them here, but they can be found in [7].

The process $P \,\hat{}\, Q$ (where $\hat{}$ is the interrupt operator) behaves like P, except that at any moment it can stop behaving like P and starting behaving like Q. This behaviour can be modelled by modifying P to offer Q as a choice at all points of its execution. Since choice distributes through most operators, so does interrupt. Therefore, distribution laws can be used to eliminate uses of the interrupt operator.

Choice, however, does not distribute through parallel composition. Implementing a process $(P_1 \parallel P_2) \,\hat{}\, Q_N$ requires offering Q_N as a choice to $P_1 \parallel P_2$, in the first place, but also stopping P_2 if P_1 is interrupted by Q_N, and vice-versa. If P involves parallel compositions we cannot translate $P \,\hat{}\, Q$ using the strategy proposed here; so we assume this situation does not happen. Apparently, the behaviour of $(P_1 \parallel P_2) \,\hat{}\, Q$ cannot, in general, be directly expressed in occam and to implement this process we would need a primitive of communication that deals with interruptions. This is one of the suggested topics for further work.

After eliminating the change of symbol, labelling, and interrupt operators, we can translate the equations to class declarations: one inner class for each process defined. We always declare a class named `Main` as follows, where P is the process named `main` in the specification.

```
private class Main implements csp.lang.Process {
    private Process mainp = P;
    public void run() { mainp.run(); }
}
```

For each process NP_i defined as P_i, we also declare a corresponding similar class. The transformation rules of Section 4.3 can be used to get from P and P_i to Java code.

Private process attributes are declared in class C to represent these processes.

```
private Main main = new Main();
private NPi npi = new NPi();
```

The run method of `C` is defined as follows: `public void run() { main.run(); }`. The run method of the `main` process is started.

A schema `enable_`c_i is translated to a method `en-ci` that returns a boolean. The components of the schema `enable_`c_i are the state components. In addition, if c_i is an input channel, the components of its schema type that are defined as simple variables are also included as components of `enable_`c_i. Therefore, if c_i is an input channel, `enable-ci` takes as argument a value of the schema type of c_i. In this case, this method is defined as follows.

```
private Boolean en-ci(Type-ci v) {
    T(Type_i) pi;
    Boolean res;

    pi = v.pi;
    res:[true,res' ⇔ enable_c_i];
    return res;
}
```

The argument `v` is an object with attributes `pi`, but `enable_`c_i refers directly to simple variables p_i. We declare these variables and assign to them the components of `v` (`pi = v.pi`).

The boolean variable `res` is used to register whether the enabling condition is satisfied or not. The command `res:[true,res'` $\Leftrightarrow$ `enable_`c_i`]` is a specification statement: it changes the variable `res`, assigning to it true if the enabling condition holds, and false otherwise. This statement can be transformed to a (sequence of) Java commands using the rules in Section 4.4.

Specification statements are common in refinement calculi; here we use the notation of [8], which is based on that of [15]. In general, a specification statement has the form $w : [pre, post]$, where w, the frame, is a list of variables, and pre, the precondition, and $post$, the postcondition, are predicates. This program can change only the value of the variables in w and, when executed from a state that satisfies pre, terminates in a state that satisfies $post$. In $post$, variables are dashed to represent their values after the execution of the statement.

If c_i is an output channel, then `enable-ci` does not need to take `v` as argument. The local variables `pi` also become unnecessary as `enable_`c_i refers only to state components.

As already mentioned, we need to extend the input mechanism of CTJ to check whether an input channel is enabled. A guard associated with an input channel can take an object of a class that implements the interface `Enable`. This object used to evaluate if a value that is available for inputting is appropriate. The interface `Enable` has only one method called `eval` which takes an an `Object` as argument and returns a boolean. This method is supposed to check whether the input value satisfies the enabling condition.

For each input channel c_i, we declare such an object as follows.

```
private Enable enable-ci = new Enable() {
    Boolean eval(Object v) {
      return en-ci((Type-ci) v);
    } }
```

We make use of a Java facility to declare and create objects of an anonymous class declared at the same time as the object, with basis on an interface. In our case, `enable-ci` is an object of a class that implements `Enable` and whose `eval` method takes an object `v` as argument and returns the result of applying `en-ci` to it. As explained, `en-ci` is defined with basis on the enabling schema associated with c_i.

If c_i is an input schema, $\texttt{effect\_}c_i$ is translated to a process. We declare a class `Effect-ci` as follows.

```
private class Effect-ci implements csp.Lang.Process {
```

The constructor for this class takes as argument a value of `Type-ci`, which is supposed to have been read from `ci`. An attribute records this value.

```
Type-ci vci;
public Effect-ci(Type-ci inp) { vci = inp; }
```

The `run` method of `Effect-ci` changes the state as specified in $\texttt{effect\_}c_i$.

```
public void run() {
    T(Type_i) pi;
    T(Type_j) pj?;

    pi = vci.pi;
    pj?  = vci.pj;
    effect_c_i;
}
```

The attribute `vci` is an object with attributes `pi`, but $\texttt{effect\_}c_i$ refers directly to simple and input variables p_i and $p_j?$. We declare these variables and assign to them the values of the corresponding components of `vci` before $\texttt{effect\_}c_i$ is executed. The schema $\texttt{effect\_}c_i$ is used in this context as a command. The transformation of this schema into Java commands is the aim of the rules discussed in Section 4.4.

When c_i is an output channel, the schema $\texttt{effect\_}c_i$ is translated to a method `effectout-ci` that returns the output specified in $\texttt{effect\_}c_i$ and to a process class `Effectst-ci` that effects the state changes. The method `effectout-ci` is as follows.

```
public Type-ci effectout-ci() {
    Type-ci res;

    [Ξ(State); p_i! : Type_i | ∃ State' • effect_c_i][p_i!\res.pi];
    return res;
}
```

The command $[\Xi(\texttt{State});\ p_i! : \mathit{Type}_i \mid \exists\, \texttt{State}' \bullet \texttt{effect\_}c_i][p_i!\backslash\texttt{res.pi}]$ is a (horizontal) schema.

The schema name `State` is used to refer to the nameless state schema in context. This command does not change the state, but produces the output specified in $\texttt{effect\_}c_i$. The local variable `res` records the output. Since $\texttt{effect\_}c_i$ refers directly to the components $p_i!$, we have to use a renaming to change occurrences of $p_i!$ with `res.pi`.

The class `Effectst-ci` is declared as shown below.

```
class Effectst-ci implements csp.Lang.Process {
```

It has no attributes and only the default constructor. Its `run` method is as follows.

$$\texttt{public void run() \{ } \exists\, p_i! : \mathcal{T}(\mathit{Type}_i) \bullet \texttt{effect\_}c_i\texttt{; \}}$$

The command $\exists\, p_i! : \mathcal{T}(\mathit{Type}_i) \bullet \texttt{effect\_}c_i$ is a schema defined using the existential quantification operator of the schema calculus. It changes the state as specified by $\texttt{effect\_}c_i$, but produces no output.

This concludes the transformation rule that applies to a CSP-OZ class. All definitions presented so far compose a single corresponding Java class. In Figure 1 we present the COZJAVA program obtained by applying our transformation strategy to the *Consumer* class presented in Section 2. The class `Type-get` used there contains only one public attribute `b` of type `Integer`. The predicate of the $\texttt{enable\_}\mathit{get}$ schema is $0 \leq b < 10$. The implementations that can be obtained for *Producer* and *Buffer* are presented in [7].

A simpler implementation of *Consumer* can be obtained if we take advantage of the fact that the schema type of the channel *get* contains only one component. In these cases, instead of defining a class that contains only one attribute, as `Type-get`, we can declare a channel that communicates values of the attribute type. In the case of the `Consumer` class, for instance, we can declare the channel attribute `get` to be of type `Channel-of-Integer`. The definitions of `enable-get` and `Effect-get` can then be simplified accordingly. For the sake of brevity, we refrain to present the rules that can be applied in this particular case, which can be found in [7].

4.2 Transformation of Class Expressions

Another form of defining a CSP-OZ class is using CSP operators to combine classes. We translate classes defined using parallel composition, non-determinisc choice, and hiding as follows. If a class C is defined as $\mathit{CExp}_1\ \mathit{op}\ \mathit{CExp}_2$, it is translated to a Java class declared as follows.

```
class C implements csp.lang.Process {
```

The class expressions CExp_1 and CExp_2 are implemented as private inner classes with fresh names `C1` and `C2`.

$$\texttt{private C1} = \mathit{CExp}_1$$
$$\texttt{private C2} = \mathit{CExp}_2$$

Further applications of this transformation rule, as well as others that follow, can be used to transform these equations to Java classes.

There is a process attribute `c` that records the behaviour of processes of class `C`.

```
private Process c;
```

This attribute is initialised by the constructor of `C` according to its definition, and using processes of class `C1` and `C2`.

The constructor method of `C` has an argument for each channel c_i in the interface of either `C1` or `C2`, and input parameter of their constructors. For each input parameter $\mathit{inp}_i?$ we declare a parameter `C1inpi?` or `C2inpi?` of appropriate type, depending on whether $\mathit{inp}_i?$ is an input parameter of the constructor of `C1` or `C2`. For each channel c_i in the interface of any of these classes, we declare a channel `ci`. If a channel c_i is in the interface of both `C1` and `C2`, then it

```
class Consumer implements csp.lang.Process {
   private Channel-of-Type-get get;
   private Integer val;

   public Consumer(Channel-of-Type-get get?)  {
      INIT;
      get = get?;  }

   private class Main implements csp.lang.Process {
      private Process mainp = (get → main);
      public void run() { mainp.run(); }
   }

   private Main main = new Main();

   public void run() { main.run(); }

   private Boolean en-get(Type-get v) {
      Integer b;
      Boolean res;

      b = v.b;
      res:  [true,res' ⇔ 0 <= b & b < 10 ];
      return res;
   }

   private Enable enable-get = new Enable() {
      Boolean eval(Object v) { return en-get((Type-get) v); }
   }

   class Effect-get implements csp.Lang.Process {
      Type-get vget;

      public Effect-get(Type-get inp) { vget = inp; }

      public void run() {
         Integer b;

         b = vget.b;
         com_get; }
   }
}
```

Figure 1: The Structure of the Implementation of *Consumer*

has the same type in both classes, as otherwise their composition would be ill-formed.

```
public C(Type-inpi?  Ciinpi?; Channel-of-Type-ci ci?)  {
```

The body of this constructor depends on whether op is the parallel or the non-deterministic choice operator. If it is the parallel composition operator, it should be as follows.

```
c = new Parallel(new Process[ ] {new C1(C1inpi,ci),new C2(C2inpi,ci)});
```

For the non-deterministic choice operator, we have the following initialisation.

```
c = new Alternative(new Guard[ ] {
    new Guard(Boolean(true),new C1(C1inpi,ci)),
    new Guard(Boolean(true),new C2(C2inpi,ci)})});
```

The `run` method is `public void run() { c.run(); }`.

Hiding can be implemented only if it is used to hide communications between two processes. For instance, if we define a class as $(C_1 \parallel C_2)\backslash c$, where c is an input channel of C_1 and an output channel of C_2, or vice-versa, in the implementation of $(C_1 \parallel C_2)\backslash c$, we regard c as a local channel. When creating an object of this class, we create this channel.

In a class definition as $C\backslash c$, where C is a class name, as opposed to a more general class expression, however, c is a global channel of C. As such, an object of C expects to communicate values through c. To make c a local channel, we would have to change the definition of C. This solution is not compositional and to avoid it we assume there are no expressions of the above form. They are useful to make global channels local and can be made unneccessary by defining the relevant channels to be local in C itself.

If C is defined as $CExp\backslash A$, using the hiding operator, it is also translated to a Java class `C`. A private inner class of fresh name `C1` is defined as `C1` $= CExp$. The constructor of `C`, however, takes as argument only channels `ci?` corresponding to the channels c_i of `C1` that are not in A (and so are not hidden).

```
public C(Type-inpi?  inpi?; Channel-of-Type-ci ci?)  {
    c = new C1(inpi?,ci?,new Channel-of-Type-cj()); }
```

The hidden channels c_j are created by the constructor.

If a class C is defined simply as `C1`, or rather, if we have an equation of the form $C =$ `C1`, again C is translated to a Java class `C` whose constructor has the same parameters as that of `C1`. It simply creates a `C1` process that is started by the `run` method.

Simple equations like these do not usually appear in CSP-OZ specifications, but can be generated by the translation rules above. We can avoid their introduction using simplified versions of these rules: if in an equation $C = CExp_1\ op\ CExp_2$ or $C = CExp_1\backslash A$ any of $CExp_1$ or $CExp_2$ is a class name, we do not need to introduce private inner classes to implement them. In the translation of C we use the names $CExp_1$ or $CExp_2$ themselves.

By way of illustration, consider the class *ProdCons* in the buffer example, which is defined as $ProdCons = ((Consumer \parallel Producer) \parallel Buffer)\backslash\{get, put\}$. We can transform it to the following class.

```
class ProdCons implements csp.lang.Process {
```

We introduce a local class PC defined as $(Consumer \parallel Producer) \parallel Buffer$.

`private PC` $= (Consumer \parallel Producer) \parallel Buffer$;

We also declare a process attribute.

```
private Process prodcons;
```

The constructor of `ProdCons` has no arguments as all channels in the interface of `PC` are hidden.

```
public ProdCons() {
   prodcons = new PC(new Channel-of-Type-get,new Channel-of-Type-put);
}
```

It initialises `prodcons` using the contructor of `PC` which, as seen below, takes a `get` and a `put` channel as parameters. These channels are created at this level.

The `run` method just starts a *PC* process.

```
public void run() { prodcons.run(); }
```

We are left now with the translation of the equation defining `PC`.

We declare this class as follows.

```
class PC implements csp.lang.Process {
```

It is defined as the parallel composition of *Consumer* || *Producer* and *Buffer*. We create a local class defined as *Consumer* || *Producer* but we do not need to introduce a local class corresponding to *Buffer*. We also declare a process attribute.

```
private CP = Consumer || Producer
private Process pc;
```

The constructor of `PC` takes the `get` and `put` channels as arguments.

```
public PC(Channel-of-Integer get, put) {
   pc = new Parallel(new Process[ ] {new CP(get,put),new Buffer(get,put)});
}
```

The `run` method runs `CP` and `Buffer` in parallel.

```
public void run() { pc.run(); }
```

The translation of the equation defining `CP` is similar.

```
class CP implements csp.lang.Process {
   private Process cp;

   public CP(Channel-of-Integer get, put) {
     cp = new Parallel(new Process[ ] {new Consumer(get),new Producer(put)});
   }

   public void run() { cp.run(); }
}
```

In this case we do no need to introduce any local classes.

The `Alternative` construct of CTJ does not support the combination of processes using general choice. We do not have a translation scheme for classes defined as $C_1 [] C_2$. Also, the `Parallel` construct is not adequate to implement interleaving. Interleaving requires independent communication of values through channels by processes that run in parallel. For example, suppose that processes P_1 and P_2 are composed by interleaving and they both read from or write to a channel c. There should be no attempt to synchronise these communications. But if P_1 and P_2 are run in parallel using the `Parallel` constructor of CTJ, synchronisation is attempted and leads to deadlock. If P_1 reads from c and P_2 writes to c, or vice-versa, then there is a proper synchronisation, but this is not the right behaviour for interleaving of processes.

It is not clear in the literature how inheritance can be used to structure the definition of the classes. There seems to be no examples available and, in particular, the combination by inheritance of the CSP parts of classes is not explained. The semantics of CSP-OZ assumes that the class hierarchy has been flattened out. As a consequence, we refrain from considering class inheritance here. A study of style and semantics is yet to be done.

4.3 Transformation of Processes

In this section we present transformation rules that can be used to translate a CSP process into Java commands that implement it.

The simplest form of process is a process name. Since there is an attribute for each named process, we can translate the process name to the corresponding attribute.

The translation of a prefixing $c_i \to P$ depends on whether c_i is a local channel or not, and if not, on whether it is an input or an output channel. If c_i is a local channel and it is enabled, then the state is changed in accordance with `effect_`c_i.

Rule 1 *Prefixing of local channel.*

$c_i \to P$

$\sqsubseteq$

```
new Sequential(new Process[ ] {new While(!enable-ci),new Effect-ci(),P})
```

provided a is a local channel.

If c_i is not a local channel, then the prefixing is translated using an `Alternative`. If it is an input channel, the translation is as follows.

Rule 2 *Prefixing of input channel.*

$c_i \to P$

$\sqsubseteq$

```
new Process() {
  Type-ci vi;
  public void run() {
    new Alternative(new Guard((ChannelInput)ci,vi,enable-ci,
        new Sequential(new Process[ ] {new Effect-ci(vi),P}))) }  }
```

provided c_i is an input channel.

We are using the new `Guard` constructor to be implemented. Since we need to declare a variable `vi` to receive the input, we create an object of a nameless class that implements the `Process` interface and has `vi` as an attribute.

For an output channel, the translation is as follows.

Rule 3 *Prefixing of output channel.*

$c_i \to P$

$\sqsubseteq$

```
new Alternative(new Guard(
   enable-ci,(ChannelOutput)ci,effectout-ci,
    new Sequential(new Process[ ] {new Effectst-ci,P})))
```

provided c_i is an output channel.

In this case we do not need to use the new `Guard` constructor and do not need to declare any extra variable.

A choice $c_1 \to P_1 \mid \ldots \mid c_n \to P_n$ is also translated using an `Alternative`. There is a guard corresponding to each $c_i \to P_i$ and the form of the guard again depends on the nature of c_i. If there are input channels, we need to create a class to declare appropriate variables to record the inputs. For a local channel c_i, the guard is as follows.

```
new Guard(enable-ci,new Sequential(new Process[ ] {new Effect-ci(),P}))
```

In this case no synchronisation on c_i is required.

We assume that in all parallel compositions $P_1 \parallel P_2$, the sets of events in which P_1 and P_2 can engage are disjoint. This assumption is reasonable: suppose that both P_1 and P_2 require a communication through a channel c. Both requirements are necessarily for inputs or outputs, so they do not synchronise. For the parallel composition to proceed, a matching input or output through c needs to occur in a third process that is run in parallel. (This process can occur in some other class specification.) However, such use of c would not respect the restriction that channels should be used for unidirectional communication between two processes only.

With the above assumption, we can translate parallel compositions using the `Parallel` constructor.

Rule 4 *Parallel.*

$P_1 \parallel \ldots \parallel P_n$

$\sqsubseteq$

```
new Parallel(new Process[ ] {P1,...,Pn})
```

Without the restriction above on the use of channels in parallel compositions, we could not translate them in this way. The process attributes of the class `C` define a context for the execution of the `main` process. This context consists of a process object `p` for each process named P in the specification. Starting from the `main` object, the thread of execution is passed from one such object to another according to the processes specifications. Parallel composition, however, splits the thread of execution.

We do not have rules to implement interleaving directly, for the reasons already explained in Section 4.1. Using laws of CSP, however, we may be able to remove interleaving from definitions.

Subordination involves the execution of a master and a slave process in parallel, where the alphabet of the slave process is a subset of that of the master. The slave process has to synchronise with the master process in all communications. For the reasons already explained, such a parallel composition does not make sense in the context of a CSP-OZ class specification.

Nondeterministic choice is implemented using `Alternative`.

Rule 5 *Nonderministic choice.*

$P_1 \sqcap \ldots \sqcap P_n$

$\sqsubseteq$

```
new Alternative(new Guard[ ]{
  new Guard(Boolean(true),P1)
  ...
  new Guard(Boolean(true),Pn)})
```

In this case the guards are all enabled.

We cannot implement general choice $P_1 [] \ldots [] P_n$, unless all the processes P_i are prefixings. The general choice $c_1 \to P_1 [] \ldots [] c_n \to P_n$ is implemented in the same way as the choice $c_1 \to P_1 \mid \ldots \mid c_n \to P_n$. The difference between these processes is that in the latter the channels c_i are supposed to be distinct, while in the former duplicated occurrences of a channel may lead to nondeterminism. This nondeterminisc behaviour is appropriately captured by the `Alternative` constructor, which nonderterministically chooses between all enabled guards. General choice distributes through most operators so that in many cases CSP laws can be applied to process definitions to write the general choices in the form contemplated by the above rule.

The process *CHAOS* is implemented as follows.

Rule 6 *Chaos.*

$$CHAOS$$

$$\sqsubseteq$$

$$\texttt{new Process()\{public void run()\{(}\mid \texttt{i} \bullet \texttt{c}_\texttt{i} \rightarrow \texttt{this)} \sqcap \texttt{STOP \}\}}$$

This is a process that can deadlock or run indefinetely offering the opportunity of communication through all the channels.

The *STOP* process is implemented as `new Alternative()`. Actually, CTJ provides an implementation for *STOP* as an infinite loop, but conceptually this implementation is more accurate. This `Alternative` deadlocks as it has no guards.

A sequential composition of processes can be translated using the `Sequential` constructor.

Rule 7 *Sequential compostion.*

$$P_1;\ \ldots;\ P_n$$

$$\sqsubseteq$$

$$\texttt{new Sequential(new Processes()}\{P_1, \ldots P_n\})$$

The process *SKIP* has a direct implementation in CTJ as `new Skip()`. In the process class `Skip`, the `run` method is empty.

Hiding of channels inside a class specification does not make sense. Hiding a global channel should not be possible and local channels are already hidden.

Changes to the state components are typically the concern of the Z part of a CSP-OZ class specification. Nonetheless, assignments, alternations, and iterations can occur in the CSP part. Their translation is very simple, as we can use the corresponding Java commands directly. For assignments, for instance, we have the following rule.

Rule 8 *Assignment.*

$$x := e$$

$$\sqsubseteq$$

$$\texttt{new Process()\{public void run()\{x = e; \}\}}$$

If $x := e$ is a multiple assignment, refinement laws of the traditional refinement calculus can be used to transform it to a standard assignment. Alternations and iterations involving conditions not directly expressible in Java also require further refinement before being translated.

4.4 Transformation of Schemas

The work reported in [8] presents rules that can be used to transform schemas to commands. These rules, with adaptations to the syntax of Object-Z and Java, can be used here. Some of them translate schemas to specification statements and other programs. Others are concerned with the rewriting of specification statements to executable programs; they are similar to the standard rules presented in works on formal development of imperative programs. These are not going to be discussed here and will be presented as needed in examples.

It is of interest to transform schemas into specification statements, because these are easier to rewrite to executable commands as they separate the pre and the postcondition of an operation.

The most general transformation rule that can be used to translate a schema is presented below. It applies to any schema $\langle \Delta(s_i);\ \Xi(s_j);\ di?;\ do! \mid p\rangle$ that specifies an operation; s_i are the state components this operation may modify, s_j are those it can refer to but not modfify, and $di?$ and $do!$ are the input and output variables, respectively.

Rule 9 *Basic conversion.*

$$\langle \Delta(s_i);\ \Xi(s_j);\ di?;\ do! \mid p\rangle$$

$$\sqsubseteq$$

$$s_i, \alpha do! : [inv \wedge \exists\, d';\ do! \bullet inv' \wedge p, inv' \wedge p]$$

where the state schema is $\langle d \mid inv\rangle$.

The frame of the specification statement includes the state components that may be changed and the output variables; for a declaration d, αd is the list of variables it introduces.

The precondition is a conjunct of pre$\langle \Delta(s_i);\ \Xi(s_j);\ di?;\ do! \mid p\rangle$. The declaration d' introduces the after-state components; it differs from d in that it introduces dashed variables s_i' and s_j', instead of undecorated variables s_i and s_j. Strictly speaking pre$\langle \Delta(s_i);\ \Xi(s_j);\ di?;\ do! \mid p\rangle$ includes the restrictions over the state and input variables introduced by d and $di?$. The method presented in the literature [18] to calculate pre$\langle \Delta(s_i);\ \Xi(s_j);\ di?;\ do! \mid p\rangle$, however, points out just $\exists\, d';\ do! \bullet inv' \wedge p$ as the precondition of the operation. When applying Rule 9, therefore, we can make use of existing precondition calculations. For any predicate p, we define p' as the predicate obtained by dashing all the free occurrences of state components in p.

As a matter of fact, CSP-OZ schemas that specify operations include, besides state components and input and output variables, simple variables. These, as already explained, have an important role in determining the enabling condition of an operation. As far as the calculation of preconditions is concerned, however, they can be regarded as ordinary input variables. The precondition of an operation defined by an (Object-)Z schema is a predicate containing only state components and input variables free. In the case of a CSP-OZ schema, the precondition may contain free simple variables as well. Nevertheless, it can be calculated in the same way.

By way of illustration, we consider the translation of the `com`_*get* operation of the class *Consumer*. Its precondition is $0 \leq b < 10$; applying Rule 9 to this schema we get the specification statement $val : [0 \leq val < 10 \wedge 0 \leq b < 10, 0 \leq val' < 10 \wedge val' = b]$. Using the Rule 10 below, we can translate it to the assignment `val = b`.

Rule 10 *Assignment introduction.*

$$w, v : [pre, post]$$

$$\sqsubseteq$$

$$v = e$$

provided$pre \Rightarrow post[e/v'][\_/']$

Syntactic Restrictions

- e is well-scoped and well-typed;
- e has no free dashed variables;
- The type of e is assignment compatible with the type ov v.

This rule, as many others, has a proviso; its application, in the above example, generates the proof obligation $0 \leq val < 10 \wedge 0 \leq b < 10 \Rightarrow 0 \leq b < 10 \wedge b = b$, which follows by reflexivity of equality and propositional calculus.

In general, transforming a specification statement will take not one rule application, but a sequence of applications of several different rules. In the above example, however, we can even apply the rule below to transform *com_get* directly to an assignment. This rule takes advantage of schemas whose predicates are an equality, or a conjunction of equalities.

Rule 11 *Assignment conversion.*

$\langle \Delta(s_i);\ \Xi(s_j);\ di?;\ do! \mid s_i' = e_i \land o_j! = e_j \rangle$

$\sqsubseteq$

$\texttt{si} = \texttt{ei};\ \texttt{oj}! = \texttt{ej}$

provided $pre \Rightarrow inv[e_i/s_i]$
where

- $o_j!$ are the output variables (elements of $\alpha do!$);
- the state schema is $\langle d \mid inv \rangle$;
- $\text{pre}\langle \Delta(s_i);\ \Xi(s_j);\ di?;\ do! \mid s_i' = e_i \land o_j! = e_j \rangle \equiv pre \land inv \land t$;
- t is the restriction introduced by d and $di?$.

Syntactic Restriction αd, $\alpha d'$, and $\alpha do!$ are not free in e_i and e_j.

The syntactic restriction guarantees that no state or output variable is free in any of the expressions e_i or e_j, and so the equalities can be established by the sequence of assignments `si = ei; oj!  = ej`.

The proviso guarantees that the state invariant is established. The predicate *pre* is supposed to be the precondition of $\langle \Delta(s_i);\ \Xi(s_j);\ di?;\ do! \mid s_i' = e_i \land o_j! = e_j \rangle$, as calculated by the method explained in the literature. Its characterisation above, however, does not uniquely determine it because conjunction is idempotent. Nevertheless, the rule provides a proper implementation of the schema if its proviso is established for any *pre* that satisfies this characterisation. This rule is a generalisation of a refinement law presented and proved in [8]; the proof of this more general rule using the theory of refinement discussed in these works is a straightforward extension of the simpler law presented there.

In our previous example, applying Rule 11 to com_*get*, we get the assignment `val = b` directly. The proof obligation is $0 \leq b < 10 \Rightarrow 0 \leq b < 10$, which is trivial.

Initialisation operations are defined by schemas named *INIT*, which take a form different from that considered in Rule 9. These schemas do not include Δ or Ξ lists, and as we assume that initialisation operations do not have inputs, they can only declare output variables. Altogether, their general form is $\langle do! \mid p \rangle$; Rule 12 can be used to translate them to a specfication statement.

Rule 12 *Initialisation conversion.*

INIT

$\sqsubseteq$

$\alpha d, \alpha do! : [\exists\, d';\ do! \bullet inv' \land p', inv' \land p']$

where

- $INIT = \langle do! \mid p \rangle$;
- the state schema is $\langle d \mid inv \rangle$.

Although *INIT* schemas do not include Δ lists, the typical purpose of these operations is to modify (initialise) the state components. Therefore, they, and the output variables, are included in the frame of the above specification statement.

The *INIT* schema of both *Producer* and *Consumer* can be translated to the specification statement below.

$$val : [\text{true}, val' = 0]$$

Using Rule 10, we can translate it to the assignment `val = 0`. Actually, using a rule similar to Rule 11, we can transform them directly to assignments.

The specification statement `res`: [true,`res'` $\Leftrightarrow$ `0 <= b & b < 10`] can be translated to the assignment `res = 0 <= b & b < 10` using Rule 10. The proof obligation is the tautology $0 \leq b < 10 \Leftrightarrow 0 \leq b < 10$. Complete implementations for all three classes: `Consumer`, `Producer`, and `Buffer`, along with a simple application class that uses them, can be found in [7].

5 Conclusions

The work reported in this paper is an intended contribution to the recent and important trend of linking theories, methods, techniques and tools for formal methods. The source of the proposed development strategy is a language, CSP-OZ, which combines two important facets of system modelling: control and data. CSP has proved to beextremely useful as a theoretical model for specification and design of the behaviour of communicating processes, whereas Z (and some of its object-oriented extensions including Object-Z) has been successfully applied both in academia and in industrial projects. As the target of our development strategy we chose Java for being a well-designed object-oriented language which has gained increasing practical acceptance, together with the CTJ package which raises the level of abstraction of concurrent programming in Java from threads to communicating processes in the style of CSP.

The similar conceptual view of process communication in the source and in the target language has, to a certain extent, made easier the proposed strategy for deriving Java programas from CSP-OZ specifications. On the other hand, there is still a big semantical gap between the communication model of both languages. In the CTJ package implemented in Java, like in CSP, a coomunication can always occur provided the relevant processes offer the syncronisation event; the syncronisation happens regardless the state of the data space of the processes. In CSP-OZ, however, even when two processes are offering a same event, say c, the communication might not happen since the (data-dependent) `enable_`c predicate might bot be satisfied. One immediate topic for further research is to redesign the communication primitives of CTJ to take this into account.

Another important aspect of the proposed strategy is that it is modular in the sense that we have a set of rules to deal with each separate concern of the development process. First we apply a somewhat complex rule which translates the CSP-OZ specification into a description with the structure of the target Java program, but still with the original CSP processes and schemas. Then we have two separate set of rules to translate the CSP operators and Z predicates into Java code. The rules for converting schemas is based on previous work [8] but the ones for the CSP operators are entirely new.

The rules, however, are not formalised. This is not a simple task: it involves a formal definition of the semantics of COZJava, of notions of refinement, and the proof that the application of the rules yield programs that refine the original specification. A formal semantics for CSP-OZ is defined in [9] using the failures-divergence model, but its extension to contemplate COZJava is a non-trivial step. A semantics for a subset of Java appropriate for the study of refinement is presented in [5, 6]. Rules very similar to those used to transform schemas are formalised in [8].

Most certainly, more efficient Java implementations of a process can be obtained if it is analysed as a whole and implemented directly, instead of using the implementations of the processes in terms of which it is defined. However, compositionality is very important to the success of transformation strategies. Furthermore, it is possible to define laws that transform the resulting programs to improve its efficiency. We are working on a refinement calculus for a Java-like language with the aiming of proposing and formalising such laws [5, 1, 2].

References

[1] P. H. M. Borba. Where are the Laws of Object-oriented Programming? In *I Brazilian Workshop on Formal Methods*, pages 59–70, Porto Alegre, Brazil, 19th–21st October 1998.

[2] P. H. M. Borba and A. C. A. Sampaio. Laws of ROOL: an object-oriented language. In *3rd Workshop on Formal Methods*, Brazil, 2000.

[3] S. M. Brien and J. E. Nicholls. Z Base Standard, Version 1.0. Technical Monograph TM-PRG-107, Oxford University Computing Laboratory, Oxford - UK, November 1992.

[4] D. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An Object-oriented Extension to Z. *Formal Description Techniques, II (FORTE'89)*, pages 281 – 296, 1990.

[5] A. L. C. Cavalcanti and D. Naumann. A Weakest Precondition Semantics for an Object-oriented Language of Refinement. In J. M. Wing, J. C. P. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1439 – 1459. Springer-Verlag, September 1999.

[6] A. L. C. CAvalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 2000. Accepted for publication.

[7] A. L. C. Cavalcanti and A. C. A. Sampaio. From CSP-OZ to Java with Processes. Technical report, Centro de Informática/UFPE, 2000.

[8] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC - A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267 – 289, 1999.

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

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[11] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In *Parallel Programming and Java Conference*, 1997.

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

[13] G. Jones and M. Goldsmith. *Programming in occam 2.* Prentice-Hall, 1988.

[14] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[15] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[16] A. C. Mota and A. C. A. Sampaio. Model-checking CSP-Z. *Science of Computer Programming*, 2000. To appear.

[17] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[18] J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof.* Prentice-Hall, 1996.