Abstract

The language CSP-OZ has been proposed as a combination of CSP and Object-Z to define state and communication aspects of concurrent systems. 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 very successful, its facilities for communication are very restricted and we make use of a library that supports the use of the concept of processes. Our work uses an existing refinement calculus for Z, but includes novel rules to deal with classes and CSP processes.

1. Introduction

With the tendency to join formalisms to obtain techniques that apply to wider classes of applications, several combinations of a process algebra [14, 16, 20] and Z [3] have been proposed. We distinguish CSP-OZ [9, 10] as a language which combines CSP and Object-Z [4] and whose syntax and semantics is very close to that of these languages. This facilitates learning and reuse of tools [11, 19].

The initial proposal of CSP-OZ is presented in [9]. In an industrial case study based on this work, we have specified and verified properties of the control system of the on-board computer of a scientific satellite [18, 19]. We have proposed a strategy to use and adapt CSP tools and model checking techniques. In [10], a new version of the language is proposed: most of the ideas are maintained, but there are syntactic and conceptual changes. We base the work reported here on the initial version of CSP-OZ, but our translation strategy and philosophy does apply to the new language.

In [10], the development of Java programs based on CSP-OZ specifications is also considered. In recognition to the high cost of formal program development, CSP-OZ specifications are used to produce assertions; the target programming language is Jass, an extension of Java with assertions. The results are not provably correct implementations, but programs enriched with assertions that can be used as a run-time device to verify correctness. Here we present rules that support a transformational approach to the development of Java [12] programs from CSP-OZ specifications, extending an existing refinement calculus for Z [8].

The wide acceptance of Java in the programming of distributed systems makes it an interesting target language. However, the facilities for concurrency of Java are based on monitors and so are not in direct correspondence with the idea of process in CSP and CSP-OZ. We chose, therefore, to use CTJ [13], a library that provides a model of processes and channels for Java based on occam [15].

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 and discuss the difficulties of implementing others. Moreover, CSP-OZ specifications separate state and communication aspects of a system. In the implementation, we have to take into account both views of the system, but deal with them independently.

We need to extend CTJ. Although the communication and process model of occam is close to that of CSP, we need a more powerful control over the communications to implement CSP-OZ specifications more directly, since the state part can interfere with and possibly block a communication.

As usual in modern development techniques [17], we use a unified language of specification and programming. It is called COZJAVA and includes CSP-OZ and Java. A command can be a Z schema, a CSP process, or a Java command. In a typical development, we start with a specification written in CSP-OZ and finish with a Java program.

In the next section we introduce CSP-OZ through an example. The CTJ library is described in Section 3. The strategy for translating CSP-OZ specifications into Java programs, is presented in Section 4. Finally, in Section 5 we summarize our results and discuss future work.
2. CSP-OZ

CSP-OZ [9] is a conservative extension of CSP and Object-Z. A process specification is similar to a class declaration in Object-Z, but extended with a clause for channel declarations and CSP equations defining the behavior of the process. The process `main` defines the control flow.

We provide an overview of CSP-OZ based on the producer/consumer example, where the producer stores information into a buffer from which the consumer retrieves. The specification of the `Producer` process is presented in Figure 1. Each channel has a record type: a schema type where there is no predicate constraining the values of the components. The channel `put` communicates records whose single component `a` is an integer. The `main` process recursively synchronizes with the environment through channel `put`.

The Object-Z part specifies the state and the operations that define the state changes and the communication effects associated with the occurrence of each CSP event. The first schema in Figure 1 defines the state of the process, which has a single component `val`: an integer between 0 and 9.

The next schema initializes the state; this initialization happens before the occurrence of any event. Each schema that defines the state change and communication effect associated with the occurrence of an event has the name of the event prefixed with `com`. The schema `com.put` determines that the communicated value through `put` is `val`, and that this value is incremented every time the event `put` occurs.

The specification of the `Consumer` process in Figure 2 is similar; it receives values through the channel `get`. Input variables can be either decorated with ? or simple (undecorated) variables, like `b` in the schema `com.get`. If restrictions imposed on the simple variables are not satisfied, the communication is blocked, but in the case of a decorated variable, the communication happens and the process diverges.

As a matter of fact, in CSP-OZ each event `c` is associated with two schemas: `enable_c` and `effect_c`. The former specifies when communications are enabled, and the latter specifies the effect on the state. By defining a `com.c` schema, we implicitly define `enable_c` as the pre-condition of `com.c`, but with the input variables existentially quantified, and `effect_c` as `com_c`. The input variables are quantified because restrictions on them do not affect the enabling of the communication. If there are no valid inputs, however, `enable_c` is false and the communication is blocked.

The `Buffer` process receives messages from the producer, through the channel `put`, and sends messages to the consumer, through channel `get`. It stores a single message at a time. For conciseness, we omit its simple definition.

Processes can be combined using CSP operators. The top level process of our system is the parallel composition of `Consumer`, `Producer`, and `Buffer`, hiding the `get` and the `put` channels so that they become internal.

\[
\text{ProdCons} = ((\text{Consumer} || \text{Producer}) || \text{Buffer})\{\text{get}, \text{put}\}
\]

We assume that process equations used inside process specifications and to combine processes are as presented in [14]. In particular, as we are interested in occam implementations, we have to assume that the channels are used for communication between two processes in only one direction.
3. CTJ

The Communicating Threads for Java (CTJ) library has been designed to raise the level of abstraction of concurrent programming in Java from threads to processes. As an example, consider the process `Producer` below.

```java
class Producer implements csp.lang.Process {
    private ChannelOutput-of-Integer put;
    public Producer(ChannelOutput-of-Integer c) {
        put = c;
    }

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

A process must implement the `csp.lang.Process` interface of the CTJ package. Processes communicate via input and output channels. These are introduced as attributes which are initialized from the parameters of the class constructor. CTJ includes interfaces `ChannelInput` and `ChannelOutput`; there are extensions of these interfaces for particular types of messages, like the `ChannelOutput-of-Integer` above. Every process must implement the method `run()`. This implementation captures the behavior of the process.

Processes can be combined in sequence, in parallel, or in alternation as in occam [15]. The synchronization is achieved by declaring channels at the network level. For example, to synchronize a producer as defined above with a consumer and a buffer process, we declare channels and use the same created instances as arguments to the constructors of the processes, as shown below.

```java
class ProdCons implements csp.lang.Process {
    Channel-of-Integer putsync = new Channel_of_Integer(),
    getsync = new Channel_of_Integer();
    Parallel prodcons = new Parallel(
        new Process[] {new Producer(putsync),
                      new Consumer( getsync),
                      new Buffer(putsync, getsync)});
    public void run() { prodcons.run(); }
}
```

The process `prodcons` is an instance of the `Parallel` class, whose constructor takes an array of processes as argument. When its method `run()` is invoked, the instances of `Producer, Consumer, and Buffer` execute in parallel.

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. Both are programs of COZJAVA and our rules are program transformations in this language. The strategy is based on a structural translation that transforms individual CSP-OZ classes to corresponding Java classes. We aim at a compositional transformation strategy.

The starting point of a program development 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.

When a COZJAVA command $c_2$ implements a COZJAVA command $c_1$, we write $c_1 \leq c_2$. 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 that still includes schemas and processes. In Section 4.2, we consider rules to transform CSP processes into CTJ processes. Section 4.3 presents rules that transform CSP-OZ class expressions into Java commands. To refine the Z schemas, we use the calculus in [8]. Only very simple adaptations to the syntax of Java are needed before we can use the laws presented in that work. These adaptations are shown in [7].

4.1. Transformation of classes

First, we consider a CSP-OZ class $C$ defined as a specification unit. It is translated to an homonymous Java class.

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

The Java class is also a process: it implements `csp.lang.Process`. In Figure 3 we present the program obtained by applying our transformation strategy to the `Consumer` class in Section 2. Implementations obtained for `Producer` and `Buffer` are presented in [7].

If $C$ includes an axiomatic description defining a constant $c$, its value $v$ to be given explicitly in an equality. This is transformed to `final static T(Type) c = v;`. The function $T$ translates a CSP-OZ type to a Java type; for instance, it translates $Z$ to `Integer`. As we assume that $C$ uses types available in Java, only, defining $T$ is not a problem.

The channels are translated to private attributes. This possibly requires the definition of subclasses of the CTJ class `Channel`. We assume that $C$ introduces a collection of channels $c_i$ that communicate bindings of their schema type. We need to define Java classes `Channel-of-Type-ci`, which implement channels of the appropriate type and is similar to the class `Channel-of-Integer` of CTJ. It uses a class `Type-ci` that implements the schema type of $c_i$ and, for instance, defines its components as public attributes. In our example, we declare the channel get of type `Channel-of-Type-get`. The class `Type-get` contains only one public attribute $b$ of type `Integer`.

Local channels are translated in the same way. They have the empty schema as their type: no value is communicated through them and they are used only as synchronization points. In CTJ, the appropriate type is `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.
class Consumer implements csp.lang.Process {
    private Channel-of-Type-get get;
    private Integer val;

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

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

    private Main main = new Main();
    public void run() {
        main.run();
    }

    private Boolean en-get(Type-get v) {
        Integer b = v.b;
        Boolean res;
        res: [true, true, res'] 0 <= b & b < 10;
        return res;
    }

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

    private classEffect-get implements csp.Lang.Process {
        Type-get vget;
        public Effect-get(Type-get inp) {
            vget = inp;
        }
        public void run() {
            Integer b = vget.b;
            get;
        }
    }

    Figure 3. The Structure of the Implementation of Consumer

    The schema \( \text{INIT} \) of \( C \), that specifies the initial state, is translated to a class constructor. Because a constructor in Java cannot return results, we assume that \( \text{INIT} \) does not have output parameters. The arguments of the constructor are the input parameters of \( \text{INIT} \) and the channels.

    \[
    \text{public } C(T\{Type\text{inp}\} inp?; \text{Channel-of-Type-ci ci}?) \{ \text{INIT}; \text{ci} = ci?; \text{cj} = new Channel-of-Any(); \}
    \]

    The schema \( \text{INIT} \) is used as a command. The channel components \( ci \) are initialized; local channels \( cj \) are created. In our example, \( \text{INIT} \) has no inputs. The constructor has one argument: a channel \( get? \) used to initialize the attribute \( get \).

    The \text{run} \ method of \( C \) is determined by the CSP part of the specification. It consists of an equation \( \text{main} = P \), which defines a process named \( \text{main} \) 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 gives rise to a private inner class declaration.

    Before introducing class declarations, however, we apply transformation rules to the equations to eliminate uses of the change of symbol, labelling, and interrupt operators, as they are not available in CTJ. The change of symbol operator amounts to a renaming of channels, and labelling can be expressed using change of symbol. The rules to eliminate the use of these operators are simple and standard [7].

    The process \( P \cdot Q \) (where \( \cdot \) is the interrupt operator) behaves like \( P \), but at any moment it can start behaving like \( Q \). This can be modeled 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 this operator. Choice, however, does not distribute through parallelism and we need a primitive of communication that deals with interruptions. This is one of our topics for further work.

    In the translation of the equations, we always declare a class named \( \text{Main} \).

    \[
    \text{private class Main implements csp.lang.Process} \{
        \text{private Process mainp = P;}
        \text{public void run() \{ mainp.run(); \}}
    \}
    \]

    The process \( P \) is that named \( \text{main} \) in the specification. A
private process attribute mainp records the behavior of main 
and the run method simply runs this process. For each pro-
cess NP, defined as Pn, we also declare a corresponding sim-
ilar class. Private process attributes are declared in class C 
to represent all these processes.

```java
private Main main = new Main();
private NPl npi = new NPl();
```

The run method of C is defined as follows.

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

In our running example, we have one equa-
tion: main = get → main. So, we declare just the class Main, 
with private attribute mainp initialized with get → main. 
The class Consumer has one process attribute: main.

A schema enable-ci; corresponds to an input channel 
ci is translated to a method en-ci that returns a boolean. 
The components of enable-ci; are those of the state and the 
simple variables the schema type of ci. The argument 
en-ci is a value of the schema type of ci. The state compo-
nents are attributes of C and can be accessed directly.

```java
private Boolean en-ci(Type-ci v) {
    T(Type-ci) pi = v.pi;
    Boolean res;
    res:[true, res' ↔ enable-ci];
    return res;
}
```

The argument v is an object with attributes pi, but 
enable-ci; refers directly to simple variables pi. We de-
clare these variables and assign to them the components 
of v. The boolean variable res registers whether the 
enabling condition is satisfied or not. The command res:[true, res' ↔ enable-ci] is a specification statement: it 
changes the variable res, assigning to it true if the enabling 
condition holds, and false otherwise. In our example in Fig-
ure 3, we define the method en-get.

If ci is an output or local channel, then we define a simi-
lar method enable-ci. In this case the argument v is not 
necessary. The local variables pi also become unnecessary 
as enable-ci; refers only to state components.

To check whether an output is enabled or not, we use 
the method enable-ci, as it has no arguments; it depends 
only on attributes that represent state components. With in-
puts, en-ci takes the input value as argument. Therefore, 
we cannot check whether the input is enabled, until it has 
actually occurred. This is, of course, not acceptable. What 
we actually do is to extend the communication mechanism 
of CTJ: even if the involved channels are ready for com-
munication, if the output value is not acceptable according to 
the enabling restrictions associated with the input channel, 
the synchronization does not occur.

This mechanism makes use of an object including a 
method eval that provides the functionality of en-ci. This 
object is an instance of a class that implements the interface

Enable, whose only method eval takes an Object as argu-
ment and returns a boolean. For each input channel ci, we 
declare such an object as follows.

```java
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 ob-
jects of an anonymous class declared at the same time as 
the object creation, with basis on an interface. In the class 
Consumer of Figure 3, we declare enable-get. Its eval 
method takes an object v as argument; it returns the result 
of applying en-get to v, which is cast to Type-get.

The translation of effect-ci also depends on whether ci 
is an input, local, or output channel. If ci is an input channel, 
effect-ci; is translated to a process class Effect-ci.

```java
private class Effect-ci implements 
csp.Lang.Process {
    Type-ci vci;
    public Effect-ci(Type-ci inp) {
        vci = inp;
    }
```

The run method of Effect-ci changes the state as specified in 
effect-ci;.

```java
    public void run() {
        T(inp) pi = vci.pi;
        T(inp) pj? = vci.pj;
        effect-ci; }
    }
```

The attribute vci is an object with attributes pi, but 
effect-ci; refers to simple and input variables pi and pj?. 
In our example, we have the class Effect-get with attribute 
vget. The argument inp of the constructor is used to ini-
italize vget. In the run method, we declare the simple variable 
b of effect-get (com_get) as a local variable and initialize 
it with vget.b.

If ci is a local channel, we also declare a class Effect-ci.
It does not have attributes, only the default constructor, and 
its run method is determined by effect-ci;.

When ci is an output channel, the schema effect-ci; is 
translated to a method effectout-ci that returns the output 
specified in effect-ci; and to a process class Effectstat-ci 
that effects the state changes.

```java
private Type-ci effectout-ci() {
    Type-ci res;
    [State; p!:+Type; State' • effect-ci][p!\res.pi];
    return res;
}
```

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

The class \( \text{Effectst-ci} \) is declared as shown below:

```java
private class \( \text{Effectst-ci} \)
    implements \( \text{csp.Lang.Process} \) {

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

```java
public void run() {\( \exists p_i! : T(\text{Type}_i) \cdot e_{ffect,c_i} \)}
```

The command \( \exists p_i! : T(\text{Type}_i) \cdot e_{ffect,c_i} \) changes the state as specified by \( e_{ffect,c_i} \), but produces no output. This concludes the transformation rule for CSP-OZ classes.

A simpler implementation of \( \text{Consumer} \) can be obtained because the schema type of the channel \( \text{get} \) contains only one component. In these cases, we can declare a channel that communicates values of the attribute type and simplify the methods. For the sake of brevity, we refrain to present the rules that can be applied in this particular case.

### 4.2. Transformation of processes

In this section we present transformation rules that can be used to translate the CSP processes in the process equations of a CSP-OZ class into Java commands. Since there is an attribute for each named process, we can process the name to that of the corresponding attribute.

The translation of a prefixing \( c_i \rightarrow P \) depends on whether \( c_i \) a local, an input, or an output channel. If \( c_i \) is a local channel and it is enabled, then the state is to be changed in accordance with \( e_{ffect,c_i} \). The corresponding process object is an instance of the class \( \text{Alternative} \). It implements the ALT construct of occam. Its constructor takes an array of guards as argument. A guard consists of a condition, an input or output through a particular channel, and a process. The process can only be chosen if the condition holds and the communication is enabled. Above, the \( \text{Alternative} \) has only one guard: its condition is \( e_{nable,c_i} \), there is no communication, as the channel is local, and the process is formed by composing an instance of \( e_{fect,c_i} \) and \( P \) in sequence.

We use an instance of another class implementing an occam constructor: \( \text{Sequential} \), which implements SEQ. Its constructor takes as argument the array of the processes to be executed in sequence. Above, if the enabling condition of \( c_i \) holds, then the state is changed as described in \( e_{fect,c_i} \), before \( P \) is executed.

For an output channel, the translation is similar. In this case, the guard of the \( \text{Alternative} \) also has \( e_{nable,c_i} \) as its condition, but there is a communication through channel \( c_i \) of the value \( e_{fectout,c_i} \). The process is formed by the sequence of an instance of \( \text{Effectst-ci} \) and \( P \).

Finally, if \( c_i \) is an input channel, we have the following.

**Rule 2** Prefixing of input channel.

\[
\text{\( c_i \rightarrow P \)}
\]

|\[
\text{new Process()} \{ \text{Type-ci vi;}
\text{public void run()} \{ \text{new Alternative(new Guard[])} \{ \text{new Guard(e_{nable,c_i},(ChannelInput)\text{ci,vi},
\text{new Sequential(new Process[]}
\{\text{new Effect-ci(vi),\text{P})})})\} run() } \}
\text{\} }}
\]

provided \( c_i \) is an input channel.

As before, we want to use an \( \text{Alternative} \) object with a unique guard. We need, however, a guard with an input communication, which requires a variable to record the input value. For this reason, the process above is not itself an instance of \( \text{Alternative} \), but an instance of a class that implements \( \text{Process} \) and whose unique attribute \( \text{vi} \) records the input value. The \( \text{run} \) method runs the \( \text{Alternative} \) process.

In the \( \text{Consumer} \) example, we have to translate the process \( \text{get} \rightarrow \text{main} \). Since \( \text{get} \) is an input channel, if we apply Rule 2, we obtain the following process.

|\[
\text{new Process()} \{ \text{Type-get vi;}
\text{public void run()} \{ \text{new Alternative(new Guard[])} \{ \text{new Guard(e_{nable-get},
\text{(ChannelInput)\text{get,vi},
\text{new Sequential(new Process[]}
\{\text{new Effect-get(vi,\text{main})})})
\text{run() } \}
\text{\} }}
\]

We use the process attribute \( \text{main} \) of \( \text{Consumer} \).

A choice \( c_1 \rightarrow P_1 \mid \ldots \mid c_n \rightarrow P_n \) is also translated using an \( \text{Alternative} \). There is a guard corresponding to each \( c_i \rightarrow P_i \), and the form of the guard again depends on the nature of \( c_i \) as shown above.

In all parallel compositions \( P_1 || P_2 \) inside a CSP-OZ class, the sets of events in which \( P_1 \) and \( P_2 \) can engage are disjoint. If \( P_1 \) and \( P_2 \) communicate through a channel \( c \), both communications are inputs or outputs and do not synchronise. For \( P_1 || P_2 \) to proceed, a matching input or output through \( c \) needs to occur in a third process that is run in parallel. However, such use of \( c \) does not respect the restriction that channels should be used for communication.
between two processes. Therefore, we can translate parallel compositions using the Parallel class.

**Rule 3 Parallel.**

\[ P_1 \parallel \ldots \parallel P_n \xrightarrow{} \text{new Parallel(new Process[]} \{P_1, \ldots, P_n\}) \]

The restriction above on the use of channels in parallel compositions is essential to the validity of this rule. A law of CSP highlights the problem: \( P \parallel P \) is not equal to \( P \). If \( P \) is, for instance, \( a \rightarrow P \parallel b \rightarrow P \), we have that \( P \parallel P = P \parallel \text{STOP} \). If we translate \( P \parallel P \) to \text{new Parallel(new Process[]} \{P, P\}), we do not have two copies of \( P \) running in parallel; \( p \) is executed in parallel with itself. The parallel composition has no effect as the choices made by the first process are necessarily exactly those made by the second process, which is the same. As a consequence of our assumption, parallel compositions like \( P \parallel P \) are not allowed.

We do not have rules to implement interleaving, because the Parallel construct is not adequate since interleaving requires independent communication of values through channels by processes that run in parallel. Using laws of CSP, however, we may remove interleaving from definitions.

Nondeterministic choice is implemented using Alternative. The condition in all guards is true and any of the processes can be chosen.

We cannot implement a general choice \( P_1 \ldots \parallel P_n \), unless all the processes \( P_i \) are prefixings. The general choice \( c_1 \rightarrow P_1 \ldots \parallel c_n \rightarrow P_n \) is implemented in the same way as the choice \( c_1 \rightarrow P_1 \ldots \parallel c_n \rightarrow P_n \). 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.

We use new Alternative() to implement \text{STOP}. CTJ provides an implementation for \text{STOP} as an infinite loop, but conceptually our proposed implementation is more accurate, since an empty Alternative deadlocks.

A sequential composition of processes can be translated using the Sequential constructor. Hiding of channels inside a class specification does not make sense. The translation of CSP assignments, conditionals, and iterations is very simple, as we can use the corresponding Java commands.

### 4.3. Transformation of class expressions

Another form of defining a CSP-OZ process is using CSP operators to combine classes. If a class \( C \) is defined as \( CExp_1 \ op \ CExp_2 \), where \( \ op \) is a parallel composition or a nondeterministic choice, it is translated to a Java class declared as follows. The class expressions \( CExp_1 \) and \( CExp_2 \) are implemented as private inner classes with fresh names.

```java
class C implements csp.lang.Process {
    private C1 = CExp1;
    private C2 = CExp2;
}
```

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

There is a process attribute \( c \) that records the behavior of processes of class \( C \).

```java
private Process c;
```

This attribute is initialized by the constructor of \( C \) according to its definition, and using processes of classes \( C1 \) 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. A channel \( c \) is in the interface of a process if it is an argument of its constructor. For each input parameter \( \text{inp}_1? \) we declare a parameter \( C1\text{inp}_1? \) or \( C2\text{inp}_1? \), depending on whether \( \text{inp}_1? \) 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 \( c_i \).

```java
public C(Type-inpi1? C1inp1;
    Channel-of-Type-ci ci1)
```

The body of this constructor depends on whether \( \text{op} \) is the parallel or the nondeterministic choice operator. If it is the parallel composition operator, it should be as follows.

```java
C = new Parallel(new Process() {
    new C1(C1inp1, ci1),
    new C2(C2inp1, ci1)});
```

For the nondeterministic choice operator, we use the Alternative class. The condition in the guards is true and none of them has a communication. For both parallelism and choice, the \text{run} method is

```java
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 \( (C1 \parallel C2)\text{\backslash} c \), where \( c \) is an input channel of \( C1 \) and an output channel of \( C2 \), or vice-versa, in the implementation, we regard \( c \) as a local channel. When creating an object of this class, we create this channel. On the other hand, in a class definition as \( C \text{\backslash} c \), \( c \) is a global channel of \( C \). To make \( c \) a hidden channel, we have to change the definition of \( C \). This solution is not compositional and we assume there are no expressions of the above form. They make global channels local and can be made unnecessary by defining the relevant channels to be local in \( C \) itself.

It is not clear in the literature how inheritance can be used to structure the definition of CSP-OZ 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.
5. Conclusions

In this work, we have proposed a development strategy for CSP-OZ with Java as a target language. We have a modular approach, in which we have a set of rules to deal with each separate concern of the development.

We use the CTJ library, which implements occam constructs. The similar conceptual view of process in the source for CSP-OZ with Java as a target language. We have a modular approach, in which we have a set of rules to deal with the relevant processes offer the synchronization event. In CSP-OZ, however, the communication might not happen since the (data-dependent) enable condition might not be satisfied. One immediate topic for further research is to redesign the communication primitives of CTJ to take this into account.

The rules presented here are not formalized. This is not a simple task: it involves a formalization of the semantics of COZJAVA, of notions of refinement, and the proof that the application of the rules yields programs that refine the original specification. A semantics for a subset of Java appropriate for the study of refinement is presented in [5, 6].

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].

Some CSP operators are not fully supported by our transformation strategy. In most cases, we can rewrite the specifications to remove them, using the CSP laws in [14]. We are, however, working on the development of a Java library that extends CTJ to include further CSP operators and support a complete transformation strategy. We are also considering a large case study in the area of protocols and plan to develop a tool to support the automatic application of our rules.

ACKNOWLEDGMENTS. The authors would like to thank Cassiane Bueno for her comments and suggestions. This work is financially supported by CNPq: grants 520763/98-0 and 521039/95-9.

References


