Unifying Classes and Processes

Ana Cavalcanti[1], Augusto Sampaio[2], and Jim Woodcock[1]

[1] University of Kent, Computing Laboratory, Canterbury CT2 7NF, England
[2] Universidade Federal de Pernambuco, Centro de Informática, P.O.Box 7851 50740-540 Recife PE, Brazil

Abstract. Previously, we presented *Circus*, an integration of Z, CSP, and Morgan's refinement calculus, with a semantics based on the unifying theories of programming. *Circus* provides a basis for development of state-rich concurrent systems; it has a formal semantics, a refinement theory, and a development strategy. The design of *Circus* is our solution to combining data and behavioural specifications. Here, we further explore this issue in the context of object-oriented features. Concretely, we present an object-oriented extension of *Circus* called *OhCircus*. We present its syntax, describe its semantics, explain the formalisation of method calls, and discuss our approach to refinement.

1 Introduction

The search for increasing levels of abstraction is a key feature in the history of Computing, and, particularly, of language design. The consolidated concepts of abstract data types and classes allow a structured modelling of real-world entities, capturing both their static and dynamic properties. The notion of process abstracts from low-level control structures, allowing a system architecture to be decomposed into cooperative and active components.

Despite the complementary nature of constructs for describing data and control behaviour, most programming languages focus only on one or the other aspect. For example, Java [18] offers (abstract) classes, interfaces, and packages; in contrast, only the low-level notion of threads is available. On the other hand, occam [21] embodies an elegant notion of process, but neglects abstract data types. There are exceptions like Ada [20], whose design has clearly addressed abstract data and control behaviour (with packages and tasks), but even so there are several limitations; for example, a package is not a first-class value.

The design of specification languages has followed a similar trend, with state-based and property-oriented formalisms concentrating on high-level data constructs [3], and process algebras exploring control mechanisms. A current and active research topic is the integration of notations to achieve the benefits of both abstract data and control behaviour [13, 31]. Our effort is *Circus* [36], a combination of Z [38] and CSP [27], which includes Dijkstra's language of guarded commands [11] and specification constructions in the style of Morgan's refinement calculus [23].

Circus is a unified language of specification, design, and programming, in the spirit of refinement calculi. In [30, 8], we define notions of refinement, and a simulation technique, which we prove sound. In [9], we propose a refinement strategy that supports the calculation of concurrent programs from centralised specifications. The semantic model of *Circus* [37] is based on the unifying theories of programming of Hoare and He [19].

In this paper, we are concerned with further investigating the integration of event and state-based formalisms, particularly within the context of elaborate state descriptions, possibly involving classes structured using inheritance and dynamic binding. Inheritance of processes is also investigated. Given our previous experience

with the development of *Circus*, our ideas are materialised by adding object-oriented features to *Circus*, and we call the new notation *OhCircus*. We provide a uniform semantic model for classes and processes, based on the unifying theories of programming.

Central to our approach is the notion and techniques of refinement: *Circus* and *OhCircus* are languages for refinement. Our aim is to provide a refinement calculus in the style of Morgan [23] that allows us to reason about concurrent object-oriented programs.

Our approach to object-orientation is also influenced in many ways by Java, but it is not our aim to stand up for its particular design. Instead, our decision is justified by our choice of target programming language: Java itself, with the support of the JCSP library [35], which implements CSP facilities. The implementation of CSP operators frees us from having to refine *OhCircus* specifications into the low-level notion of Java threads, which involves shared variables, rather than communication via channels.

Our case studies have shown that this framework is suitable for the implementation of *Circus* specifications, and it is of practical interest due to the success of Java [26]. By including object-oriented facilities in *Circus*, we hope to be able to better explore the facilities of Java.

In the next section, we present and justify our approach to modelling states and events. In Section 3, we introduce *OhCircus* and provide a small example to illustrate the combined use of classes and processes. Section 4 describes the semantics of method calls. The following section discusses a unified strategy for refinement of processes and classes. Finally, we discuss our results and topics for further research.

2 The *Circus* approach

The structure of a *Circus* program is similar to that of a Z specification; a program in *Circus* is a sequence of paragraphs: Z paragraphs, process definitions, or channel definitions. Typically, a system is modelled as a process.

The most basic way to define a process is by explicitly specifying its state, using a schema as in Z, and its behaviour, using a (main) action. Schemas that define operations over the state are actions; furthermore, constructors of CSP and of Morgan's refinement calculus can also be used to define actions. A main action is distinguished as the definition of the overall behaviour of the process. A process can also be defined in terms of existing processes using CSP operators.

As a very simple example, we consider a process that models a buffer with two positions; it uses channels *in* and *out* to input and output integer numbers.

$$\textbf{channel } in, out : \mathbb{Z}$$

We call our example process *TwoPositionBuffer*.

$$\textbf{process } TwoPositionBuffer \mathrel{\widehat=} \textbf{begin}$$

The state of the buffer contains components *first* and *second* to hold the buffered numbers, and a component *size* that determines how many numbers are actually in the buffer.

state *BState*

$$first, second : \mathbb{Z}$$
$$size : \mathbb{N}$$
$$size \leq 2$$

The invariant states that at most two integers are buffered.

An initialisation schema defines an action that sets the size of the buffer to 0: initially, the buffer is empty.

Init

$$BState'$$

$$size' = 0$$

This schema is not distinguished from the others. The behaviour of a process does not necessarily start with an initialisation of the state. Instead, it is defined by the distinguished main action, which may perform an initialisation; this is the case in our example, as shown below. In general, operations over the state are executed only when invoked from the main action of the process.

The first operation that we present inserts a number $x?$ into the empty buffer.

InsertFirst

$$\Delta BState$$
$$x? : \mathbb{Z}$$

$$size = 0$$
$$first' = x?$$
$$size' = 1$$

The next operation inserts a second number into a buffer that already holds one number. The first number is not changed.

InsertSecond

$$\Delta BState$$
$$x? : \mathbb{Z}$$

$$size = 1$$
$$first' = first$$
$$second' = x?$$
$$size' = 2$$

The output of a number is possible only if the buffer is not empty, in which case the output number is always *first*. The operation *Output* updates the state accordingly, but does not actually produce the output; this is left for the main action.

Output

$$\Delta BState$$

$$size > 0$$
$$first' = second$$
$$size' = size - 1$$

If the buffer happened to have two numbers, the *second* number becomes the *first*, and is output in the next request.

All these state operations are auxiliary actions that we use below to specify the main action of *TwoPositionBuffer* using CSP operators.

$$
\begin{array}{l}
\bullet\ \mathit{Init}; \\
\quad \mu\, X \bullet (\mathit{size} < 2)\ \&\ \mathit{in}?x \rightarrow \\
\qquad\qquad (\mathit{size} = 0)\ \&\ \mathit{InsertFirst};\ X \\
\qquad\qquad \Box \\
\qquad\qquad (\mathit{size} = 1)\ \&\ \mathit{InsertSecond};\ X \\
\qquad\quad \Box \\
\qquad\quad (\mathit{size} > 0)\ \&\ \mathit{out}!\mathit{first} \rightarrow \mathit{Output};\ X \\
\textbf{end}
\end{array}
$$

First of all, the state is initialised. In sequence, the process *TwoPositionBuffer* recurses (μ), offering the possibility of inputting and outputting numbers depending on its *size*. If the buffer is not full, $\mathit{size} < 2$, then it is possible to input a number x. An action $g\ \&\ A$ behaves like A if g holds, and deadlocks otherwise. For an action $c \rightarrow A$, the behaviour is that of A, after the communication c. The communication $\mathit{in}?x$ is the input of a value x through the channel *in*. If the buffer is not empty, then it is also possible to output *first*; the communication $\mathit{out}!\mathit{first}$ outputs the value *first* through the channel *out*. In the main action, $\mathit{size} < 2$ and $\mathit{size} > 0$ are used as guards, as are $\mathit{size} = 0$ and $\mathit{size} = 1$. After an input, the action *InsertFirst* or *InsertSecond* is executed depending on whether the buffer is empty or not.

Events, which in *Circus* are communications, and state updates are detached. In our example, the occurrence of the communication $\mathit{in}?x$, for instance, does not trigger a state update. The following action is an external choice ($\Box$); depending on the value of *size*, either *InsertFirst* or *InsertSecond* is going to be used to update the state.

As opposed to other integrated formalisms, *Circus* does not identify guards and preconditions, which have different purposes. The precondition of *InsertFirst*, for example, is that the buffer is empty, $\mathit{size} = 0$. If executed in a state in which *size* is 1 or 2, then *InsertFirst* diverges. In this way, the behaviour of an action defined by a schema operation is the same as in Z; as a consequence, refinement of schemas is as expected by Z users. The main action uses $\mathit{size} = 0$ as a guard to block the choice to execute *InsertFirst*, if its precondition does not hold.

In spite of the great flexibility provided by *Circus*, it is possible to start describing a system either exclusively with events or exclusively with states. A process may have no state components, and have its behaviour defined by a main action that uses only CSP constructs. Such a process is described using a process algebra, with no explicit state. Later, we can apply data refinement to introduce a state.

As a second option, it is possible to focus on an abstract data type as a system specification, and later develop the communication and concurrency required for a distributed implementation. This amounts to writing a Z specification, and using channels to communicate the inputs and outputs of each operation, because the state of a process is local. The main action initialises the state and recursively offers the choice of all operations. For example, a process that models an integer

variable can be specified as follows.

$$
\begin{array}{l}
\textbf{channel}\ rd, wrt : \mathbb{Z} \\
\textbf{process}\ Variable \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat=} [\, x : \mathbb{Z} \,] \\
\quad Init \mathrel{\widehat=} [\, State' \mid x' = 0 \,] \\
\quad Read \mathrel{\widehat=} [\, \Xi State;\ v! : \mathbb{Z} \mid v! = x \,] \\
\quad Write \mathrel{\widehat=} [\, \Delta State;\ v? : \mathbb{Z} \mid x' = v? \,] \\
\quad \bullet\ Init; \\
\qquad \mu\, X \bullet \textbf{var}\ v : \mathbb{Z} \bullet Read;\ rd!v \rightarrow X \\
\qquad\qquad \Box \\
\qquad\qquad wrt?v \rightarrow Write;\ X \\
\textbf{end}
\end{array}
$$

The state component is the value x of the variable. There are operations *Init* to initialise the state, and *Read* and *Write* to access and update the variable, which are defined just as in Z; in *Circus* they are actions. The action *Read* has an output variable $v!$, which we declare locally as v. After *Read* is executed, we output v through the channel rd.

The channels rd and wrt are needed because the state of a *Variable* process is encapsulated; the only way of interacting with a process is through channel communication. A process that does not input or output any information is equivalent to *Skip*, if it terminates, or is divergent, if it does not, no matter how complex its state is. If this pure Z style is adopted, the *Circus* refinement strategy can be used to introduce a rich system architecture [9].

In summary, even if any of the pure specification styles is adopted, refinement allows moving to a hybrid description during development. Typically, in code, there is a complex interplay between events and state updates, just as in programs written in languages like Java and occam. Even though it is possible, moving between the pure styles of specification is not the objective when you have coding in mind.

Most often, it is more convenient to adopt a mixture of the two approaches from the early stages of specification, using parallelism to capture requirements separately, and using states to abbreviate and simplify descriptions. Such *Circus* specifications make an important distinction between the use of events and states as specification devices and as implementation artifacts.

A temptation that arises from the availability of both Z and CSP constructs is to introduce too much structure into the specifications. Parallelism, for example, is useful to combine the components of a design; at the abstract specification level, it should be used only when there is intrinsic parallelism in the requirements.

In most cases, an abstract *Circus* specification is composed of a single process whose definition is structured using the schema calculus and action operators. Global properties should be captured as invariants in the style of Z. As we progress with refinement, these properties are distributed throughout the components and become local invariants interconnected by reactive mechanisms.

Another approach to the combination of CSP and Z is represented by CSP-Z [14]. The major distinction between CSP-Z and *Circus* is that, in CSP-Z, reactive behaviour is defined solely using CSP, with an implicit attachment between an event in the CSP description and a state transition described as an associated Z operation. In other words, CSP-Z embeds the second specification style we considered above.

The work in [31] presents CSP||B, a combination of CSP and B [1] that follows the philosophy of communicating data types. In this work, CSP is used to control interaction between B machines; a B machine is regarded as a data type, with each operation triggered by an associated communication.

The fixed architectural model adopted in both CSP-Z and CSP||B for the specification of data and behavioural aspects of the system has advantages. Firstly, in

the resulting specifications there is a clear separation between the uses of the combined notations. This allows the easy reuse of successful existing tools like FDR [16]. Secondly, the semantics of the language can be defined as an extension of the well-studied failures-divergences semantics of CSP, with the view of the data types as processes.

The motivation for the design of *Circus* and *OhCircus*, however, is the definition of a unified language for refinement. In this context, the detachment of an event occurrence from a state transition seems more appropriate. Potential target programming languages like Java, occam, or Handel-C [33] also deal with events and state update independently.

As already said, the semantics of *Circus* and *OhCircus* is based on the unifying theories of programming. As such, even though it is not a direct extension of the semantics of CSP, it reuses an existing model that encompasses state-based, reactive, and concurrent constructs. Furthermore, we are currently working on the development of a model-checker for *Circus* based on FDR, and on the mechanisation of the semantics of *Circus* using a Z theorem prover: ProofPowerZ.

Our approach to model checking *Circus* specifications combines standard model checking techniques with theorem proving. We generalise the successful algorithm of FDR to handle infinite automata that may arise from the rich state of *Circus* specifications. Failures and divergences are handled symbolically, and the outcome of a model checking attempt depends on a set of proof obligations that need to be discharged using a theorem prover.

Typically, model checking is used to debug specifications and implementations through a cycle of checks and amendments. In our approach, an application-oriented theory is developed for a particular specification during the cycles. As we progress, this theory should become powerful enough to support automatic theorem proving.

Handling infinite and complex data structures is also an issue when model checking CSP-Z and CSP||B specifications. In the case of CSP-Z [25, 12], data abstraction is used. In the CSP||B approach, relevant data properties of the B machines are recorded as assertions in the CSP processes.

In [10], refinement of Object-Z [32] specifications is studied in detail, and a possible combination of Object-Z and CSP is also considered. In Object-Z, the precondition of the Z operation schemas are taken as guards for their execution. In *Circus*, we keep the philosophy of Z and reflect the generality of programming by introducing a guard constructor. This has a major impact on refinement, so that the rules that we need for *Circus* (and for *OhCircus*) are different, and are closely related to those of Z.

The notation CSP-OZ [14] follows design guidelines similar to those of CSP-Z, but, being a combination of CSP and Object-Z [32], it also includes classes, inheritance, and other object-oriented features. Besides coupling events and state transitions, classes and processes are identified in CSP-OZ. There, the main concern is specification only, and at that level of abstraction such distinctions may not be really necessary.

In industrial practice, however, class instances (objects) are passive entities, while processes are active. It seems artificial to force class descriptions that model, for instance, addresses or employees, to have the status of a process, since the standard behaviour would be just a recursive choice of all its operations (methods). Another potential practical inconvenience of regarding data elements as processes is that method calls have to be expressed as synchronisations involving explicit communication.

In *OhCircus*, where refinement is a central issue, we introduce a separate construct for defining classes, and provide a semantics for method calls that is independent of communication. Classes support the definition of processes with complex state spaces. Their introduction raises issues related to the refinement of classes,

but does not interfere with existing results related to the development of processes. When deriving from an *OhCircus* specification code written, for example, using JCSP, it will usually be relevant to distinguish between ordinary classes and active ones (processes).

The work reported in [22] is an integration of Object-Z and Timed CSP called TCOZ. Like *Circus* and *OhCircus*, TCOZ does not follow the communicating data type philosophy, and avoids implicit associations between communications and state updates. Their motivation is the need to specify time restrictions on state operations; if they are identified with communications, they become instantaneous in the framework of CSP.

Refinement does not seem to be a concern in that line of work. Indeed, TCOZ inherits the difficulties related to refinement of Object-Z specifications. For instance, in Object-Z, method calls can be used as schemas; since the schema calculus operators are not monotonic with respect to refinement, stepwise refinement of methods is not immediately available. Furthermore, in Object-Z class inheritance allows renaming, redefinition, and cancellation of operations in such a way that a subclass is not necessarily a refinement or even a subtype of its superclasses.

In *OhCircus*, the use of schemas, method calls, and the subclassing mechanism is restricted, so that modular reasoning and stepwise refinement are still possible. This may lead to more verbose specifications of redefined operations than those written in CSP-OZ or Object-Z; examples are presented in the next section. It is our view, however, that the loss of a compositional approach to refinement would be too high a price to pay. Furthermore, the use of the unifying theories of programming as a basis for the *OhCircus* model makes the use of method calls in expressions and predicates possible, with great improvement in expressive power.

Another issue is that of copy versus reference semantics. *Circus* and *OhCircus* do not allow sharing. The refinement of a specification into target languages like JCSP [26] or UML-RT [29] preserves the channel based communication model for processes. Object-oriented program development exploring the use of patterns, possibly involving sharing, is a suggested topic for further research. Results in the absence of sharing are reported in [4]. Object-Z and CSP-OZ have a reference semantics. Nevertheless, as already pointed out, there seems to be no proposal of how to formally refine specifications in those languages into object-oriented programs. The next section details the rationale for the design of *OhCircus*.

3 *OhCircus*

In the same way as *Circus*, and Z, a program in *OhCircus* is a sequence of paragraphs. In *Circus* they can be Z paragraphs, or process and channel definitions. *OhCircus* includes yet another form of paragraph: a class definition. Moreover, in *OhCircus* we can define processes using inheritance, and using data types defined as classes.

In Figures 1 and 2, we provide a partial BNF description for the syntax of *OhCircus*. The syntactic categories Paragraph, Schema-Exp, Expression, and Declaration contain Z paragraphs, schema expressions, expressions, and declarations. The syntactic categories named ChannelDefinition, ChanSetDefinition, and CSExpression are those of *Circus* channel definitions, channel set definitions, and channel set expressions. Finally, the syntactic category N contains the valid names. Terms included in brackets are optional. Superscript + is used for non-empty comma-separated lists of elements of the base syntactic category; if the list can be empty, we use superscript *.

A process definition in *OhCircus* gives the process name and description, as in *Circus*, but can also include an **extends** clause, which names a superprocess. The defined process is said to be a subprocess of that named in the **extends** clause.

Program	::=	OhCircusParagraph*	
OhCircusParagraph	::=	Paragraph	
	|	ChannelDefinition | ChanSetDefinition	
	|	OhProcessDefinition | ClassDefinition	
OhProcessDefinition	::=	**process** N $\widehat{=}$ [**extends** N] Process	
Process	::=	**begin**	
		PParagraph*	
		[**state** Schema-Exp | Constraint]	
		PParagraph*	
		[• Action]	
		end	
	|	N | Process; Process	
	|	Process □ Process | Process ⊓ Process	
	|	Process ⟦ CSExpression ⟧ Process	
	|	Process ||| Process | Process \ CSExpression	
OhExpression	::=	Expression	
	|	**this** | **null**	
	|	**new** N[(OhExpression$^+$)]	*constructor*
	|	OhExpression.N	*attribute selection*
	|	OhExpression.N(OhExpression*)	*method call*
	|	**super**.N	*attribute selection*
	|	**super**.N(OhExpression*)	**super** *call*
	|	OhExpression **instanceof** N	*type test*
	|	(N)OhExpression	*type cast*

Fig. 1. *OhCircus* syntax

The state of a subprocess includes all the components of its superprocess. In a subprocess specification, we have access to all the definitions of the superprocess: state components, actions, and auxiliary definitions. Furthermore, the main action of the subprocess is implicitly composed in parallel with the main action of the superprocess. The reason for this is the result on behavioural refinement of processes presented in [15, 34].

When we declare a subprocess in *OhCircus*, we raise a proof obligation to show that the subprocess behaviourally refines its superprocess. In [15], the authors put forward a behavioural refinement relation that guarantees substitutability in all contexts. This means that, if a process P_1 is behaviourally refined by another process P_2, according to their definition, then uses of P_1 can be replaced with uses of P_2 in all contexts. Furthermore, in [34], it is indicated that, having the behaviour of P_2 defined by the parallel composition of that of P_1 with an additional process is part of a set of conditions which, together, are sufficient for attaining behavioural refinement. Therefore, even though this is not enough, by considering that the main actions of the superprocess and the subprocess are in parallel, we are taking a first step towards behavioural refinement.

Parallelism of processes is just like in CSP: the parallel processes synchronise on communications through channels in a given synchronisation set. The states of the processes are encapsulated and are handled independently. For actions, we need to identify partitions of the process state to avoid conflict. For example, the action $A_1 \llbracket \{ x \} \mid \{\!| c |\!\} \mid \{ y \} \rrbracket A_2$ is the parallel composition of A_1 and A_2, synchronising on communications through c; both A_1 and A_2 have access to the state components x and y, but A_1 can only change x and A_2 can only change y. In the case of the main action of a subprocess $SubP$, which is implicitly defined using parallelism, the

ClassDefinition	::=	**class** N $\widehat{=}$ [**extends** N] **begin** CParagraph* [**state** StateSchema | Constraint] CParagraph* [**initial** Schema-Exp] CParagraph* **end**
CParagraph	::=	Paragraph | Qualifier N $\widehat{=}$ ParametrisedCommand
Qualifier	::=	**public** | **protected** | **private** | **logical**
ParametrisedCommand	::=	Schema-Exp | Command
	|	ParameterDeclaration • Command
ParameterDeclaration	::=	ParameterQualifier Declaration
	|	ParameterQualifier Declaration; ParameterDeclaration
ParameterQualifier	::=	**val** | **res**
Command	::=	N$^+$: [Predicate, Predicate] | N$^+$:= Expression$^+$
	|	OhExpression.N(OhExpression*) | **super**.N(OhExpression*)
	|	Command; Command | μ N • Command
	|	**if** GuardedCommands **fi** | **var** Declaration • Command
GuardedCommands	::=	Predicate $\rightarrow$ Command
	|	Predicate $\rightarrow$ Command $\Box$ GuardedCommands

Fig. 2. *OhCircus* syntax

main action of the superprocess *SuperP* changes its state components, and the main action defined in *SubP* changes only the additional components.

An explicit process description is like in *Circus*: a sequence of paragraphs and a main action. The main action is optional in *OhCircus*, and, if omitted, it is assumed to be *Skip*; this is not usually an interesting main action for a *Circus* process, but it is perfectly reasonable for an *OhCircus* subprocess. In this case, the main action of the subprocess is the parallel composition of the main action A of the superprocess with *Skip*: simply A. In effect, if the main action of a subprocess is omitted, that of the superprocess is inherited as it is. All the *Circus* process operators are available, but, for the sake of simplicity, we include only some in Figure 1.

The syntactic category PParagraph is that of *Circus* process paragraphs. We extend only the expression notation to include several object-oriented constructs: **this**, **null**, and the **new** constructor, all in the style of Java, attribute access and method calls, possibly with target **super** as in Java, and type tests and casts.

The syntax of class definitions is depicted in Figure 2. It gives a class name, its superclass in an **extends** clause, and its attributes and methods delimited by **begin** and **end**. If the optional **extends** clause is omitted, the superclass is **object**, which has no attributes or methods. As with subprocesses, the declaration of a subclass raises a proof obligation to guarantee that it behaviourally refines the superclass.

To illustrate our notation, we present a bank system inspired by that defined in [10] using Object-Z. First of all, we have a class *Account* to model individual bank accounts. In its specification, we use a given set *NUMBER* containing all valid account numbers.

class *Account* $\widehat{=}$ **begin**

The definition of the attributes and methods of a class consists basically of a sequence of paragraphs. An optional **state** clause distinguishes a schema expression or a constraint that defines the state of the class; we sometimes refer to the state components as attributes. If the state definition is just a constraint, the state com-

ponents are those of the immediate superclass; the constraint strengthens the invariant.

The state schema expression may be just as in Z, but it may also include qualifiers in the declaration of its components. If nothing is said, the attributes are private, but we may declare them to be protected or public. Protected attributes are visible in the subclasses, and public attributes are visible in the whole program. For processes, we do not have this possibility: the state components are always private, as interaction with a process is possible only through channels.

In our example, the state schema declares two protected attributes: the *number* of the account and its *balance*.

state *AcctState*

$$\textbf{protected}\ number : NUMBER$$
$$\textbf{protected}\ balance : \mathbb{Z}$$

The **initial** clause introduces a constructor. It defines the meaning of **new**. The initialisation schema cannot have outputs and must be a deterministic operation, to avoid the complexities of nondeterministic expressions [24]. In the bank, the initialisation takes the number of the account as input.

initial *AcctInit*

$$AcctState'$$
$$number? : NUMBER$$

$$number' = number?$$
$$balance' = 0$$

The balance is set to 0 initially.
The methods are distinguished from other paragraphs because they are qualified as private, protected, public, or logical. The definition of a method is a parametrised command: a schema expression, a command itself, if there are no parameters, or a parametrised command in the style of Back [2]. Below, we have a public method *Deposit* defined as a schema.

public *Deposit*

$$\Delta AcctState$$
$$amount? : \mathbb{N}$$

$$balance' = balance + amount?$$
$$number' = number$$

This method takes the *amount*? to be deposited as input, and updates the balance accordingly. The definition is just as in Z, or *Circus*.

The *Withdraw* method is defined as a parametrised command in the style of Back. Parameters can be passed by value or result. Here, we have a value parameter *amount*; the body of the parametrised command is a specification statement.

$$\textbf{public}\ Withdraw \mathrel{\widehat=} \textbf{val}\ amount : \mathbb{N} \bullet balance : \left[\begin{array}{l} amount \leq balance, \\ balance' = balance - amount \end{array} \right]$$

The specification describes a program that may change the attribute *balance*. The precondition requires that the parameter is an amount available in the account. In this case, the postcondition defines that the final balance of the account is obtained by deducting the amount taken as input.

Finally, the method *Balance* has a result parameter; its body is an assignment.

$$\textbf{public}\ Balance \mathrel{\widehat=} \textbf{res}\ bal : \mathbb{Z} \bullet bal := balance$$

In the definition of a method as a schema, the input variables are parameters passed by value, and the output variables are parameters passed by result.

Methods qualified as logical are specification artifacts; they are useful to describe a specification or design, but do not need to be implemented in the final code. Below, we define a logical method *getNumber*, which returns the *number* of the account through a result parameter n; it is used in the specification of the bank.

$$\begin{array}{l}\textbf{logical}\ getNumber \mathrel{\widehat=} \textbf{res}\ n : NUMBER \bullet \\ \quad n := number \\ \textbf{end}\end{array}$$

Strictly speaking, we do not really need *getNumber*. In a specification or design, the visibility of attributes and methods does not need to be respected. In a class specification, if an attribute of another class is present, direct access to its attributes may be useful to describe invariants and methods. For example, in the specification of the bank, we can directly access the *number* and *balance* attributes of the accounts. Logical methods, however, are useful as a structuring mechanism when the information that needs to be accessed is not directly available as an attribute. In fact, direct accesses to attributes can be viewed as implicit uses of logical get and set methods. In code, visibility constraints must be respected.

An account that provides some credit can be described as a subclass of *Account*.

$$\textbf{class}\ CreditAccount \mathrel{\widehat=} \textbf{extends}\ Account\ \textbf{begin}$$

The state of a subclass includes all the attributes of its (direct) superclass. More specifically, the state schema of the superclass is implicitly included in that of the subclass. This allows the definition of an invariant that involves the components of both states. In our example, we have one extra attribute: the *credit*.

state *CAcctState*

$$\textbf{protected}\ credit : \mathbb{N}$$

$$balance + credit \geq 0$$

The invariant guarantees that the credit is never exceeded; it involves attributes of both *Account* and *CreditAccount*.

The constructor of *CreditAccount* takes the initial credit as input.

$$\begin{array}{l}\textbf{initial}\ CAcctInit \mathrel{\widehat=} \\ \quad \textbf{val}\ number : NUMBER;\ initCred : \mathbb{N} \bullet \\ \qquad AcctInit;\ credit := initCred\end{array}$$

We observe that *AcctInit* is used as a command in this context. Even though *AcctInit* is a constructor, it can be used as a command in the constructor of the subclass.

Perhaps, a definition that would spring to mind more readily would be the following.

initial *CAcctInit*

$$CAcctState'$$
$$AcctInit$$
$$initCred? : \mathbb{N}$$

$$credit' = initCred?$$

This, however, is not valid in *OhCircus*, since methods (and constructors) cannot be used as schemas, even if they are defined by a schema. This is because, as already said, in general, the schema calculus operators are not monotonic with respect to refinement. So, if we refined a method, all its uses as a schema would potentially need to be modified, and certainly would need to be checked. This approach is not practical. Programming constructors like sequence, on the other hand, are all monotonic. If we refine *AcctInit*, for example, we are also refining *CAcctInit*.

If a method is redefined, there is also no implicit inclusion of the schema that defines it in the superclass. There is actually no guarantee that it is defined by a schema in the superclass; even if it is, refinement may transform it into another construct. If an implicit inclusion were assumed, refinement would require changes in subclasses.

Methods that are defined by a schema in the superclass, and are not redefined, are extended to the larger state, implicitly. They are conjoined with a schema that specifies that the attributes of the subclass do not change. For example, we do not redefine *Deposit*, therefore, it is available in this class with the definition below.

public *Deposit*

$$\Delta CAcctState$$
$$amount? : \mathbb{N}$$
$$balance' = balance + amount?$$
$$number' = number$$
$$credit' = credit$$

The inclusion of $\Delta CAcctState$ puts the attributes in context.

We redefine the method *Withdraw*; in the case of a credit account, more can be withdrawn than simply the balance of the account. The credit is also available.

$$\textbf{public}\ Withdraw \mathrel{\widehat=} \textbf{val}\ amount : \mathbb{N} \bullet balance : \left[\begin{array}{l} amount \leq balance + credit, \\ balance' = balance - amount \end{array} \right]$$

This definition of *Withdraw* is a refinement of that in the superclass *Account*: its precondition is weaker. A method redefinition must not change its signature: it needs to have the same parameters, with the same type.

Finally, we also have a new method, which sets the credit of the account.

public *SetCredit*

$$\Delta CAcctState$$
$$\Xi AcctState$$
$$credit? : \mathbb{N}$$
$$credit' = credit?$$

end

We use the schema $\Xi AcctState$ to concisely define that the attributes of the superclass are not affected by this method.

The bank is modelled as a process. Four channels are declared: *open*, *deposit*, and *balance* are used to request that an account is open, that a deposit is made, and the balance of an account, and *out* is used to receive balance information.

$$\textbf{channel}\ open : Account;\ balance : NUMBER;\ deposit : NUMBER \times \mathbb{N};\ out : \mathbb{N}$$

In our very simple example, we omit several operations.
The process *Bank* is a client of the classes *Account* and *CreditAccount*. Its state includes a set of accounts, which can be *Account* or *CreditAccount* objects.

process $Bank \mathrel{\widehat=}$ **begin**

The state of the bank has one attribute: a set of *accounts*.

state *BState*

$$accounts : \mathbb{P}\, Account$$

$$\forall\, acct : accounts \bullet acct \neq \textbf{null}$$
$$\forall\, acct_1, acct_2 : accounts \mid acct_1 \neq acct_2 \bullet$$
$$\quad acct_1.getNumber() \neq acct_2.getNumber()$$

The invariant guarantees that all accounts are proper instances of *Account*: they are not **null**, and they have different numbers. The (logical) method *getNumber* of class *Account* is used in this definition to access the *number* attribute of the accounts. Calls to methods like *getNumber*, which are deterministic and have exactly one result parameter, can be used as values. Such use of a method, however, generates a proof obligation to guarantee that it is indeed deterministic. In the case of *getNumber*, this is trivial.

Method calls cannot be used in a negative context, like a negation or the antecedent of an implication, otherwise the resulting predicate is not monotonic. This restriction does not cause too many difficulties in *OhCircus*, since we are free to use the names of any attributes directly in a predicate, regardless of visibility constraints. As we said before, visibility constraints apply to program code, but not necessarily to specifications and designs. For example, we can specify the state of the bank as follows.

state *BState*

$$accounts : \mathbb{P}\, Account$$

$$\forall\, acct : accounts \bullet acct \neq \textbf{null}$$
$$\forall\, acct_1, acct_2 : accounts \mid acct_1 \neq acct_2 \bullet$$
$$\quad acct_1.number \neq acct_2.number$$

In this case, we access the attribute *number* of the accounts directly. The choice of approach is a matter of style.

The initialisation is very simple. The collection of accounts is initially empty.

$$BInit \mathrel{\widehat=} [\, BState' \mid accounts' = \emptyset \,]$$

Since *Bank* is a process, and not a class, this is not a constructor, but an action.

To open a bank account, we give an account as input.

Open

$$\Delta BState$$
$$acct? : Account$$

$$acct? \neq \textbf{null}$$
$$\forall\, acct : account \bullet$$
$$\quad acct.getNumber() \neq acct?.getNumber()$$
$$accounts' = accounts \cup \{\, acct? \,\}$$

Some actions are defined in terms methods of *Account* using the Z promotion technique. We use a promotion schema that we call *Lookup*.

Lookup

$\Delta BState$
$number? : NUMBER$
$acct, acct' : Account$

$acct \in accounts$
$acct.getNumber() = number?$
$accounts' = (accounts \setminus \{ acct \}) \cup \{ acct' \}$

This schema provides a frame for operations that act over an existing account *acct*, and produce a modified account $acct'$. It identifies *acct* as the account with the number given as input; it also updates *accounts* by removing *acct* and inserting the updated $acct'$.

Apart from the account number, the *PDeposit* action also takes the $amount?$ to be deposited as input. Its specification is a call to the *Deposit* method of *Account* with target *acct*.

private *PDeposit*

Lookup
$amount? : \mathbb{N}$

$acct.Deposit(amount?)$

The method call $acct.Deposit(amount?)$ denotes the predicate below, which specifies the effect of depositing $amount?$ in the account *acct*.

$$
\begin{aligned}
&acct'.balance = acct.balance + amount? \land \\
&acct'.number = acct.number \land \\
&acct'.credit = acct.credit
\end{aligned}
$$

This predicate can be calculated from the specification of *Deposit*, as explained in the next section. It is a predicate over *acct* and $acct'$, which are both in scope. The possibility of using method calls in predicates accounts for clearer and more concise specifications.

The components *acct* and $acct'$ need to be hidden, as they are used only to promote the call to *Deposit*.

$$Deposit \mathrel{\widehat=} PDeposit \setminus (acct, acct')$$

For conciseness, we do not consider *Withdraw*, which can be promoted in a similar way. *Balance* is as follows.

$$
\begin{aligned}
&PBalance \mathrel{\widehat=} [Lookup;\ bal! : \mathbb{Z} \mid acct.Balance(bal!)] \\
&Balance \mathrel{\widehat=} PBalance \setminus (acct, acct')
\end{aligned}
$$

Like *PDeposit*, the schema *PBalance* includes *Lookup* and declares a new component; in this case, a result parameter $bal!$. The result parameter of the account method *Balance* defines the value of $bal!$.

The main action below defines the behaviour of the bank.

$$
\begin{array}{l}
\bullet\ \mathit{BInit}; \\
\quad \mathit{open}?\mathit{acct} \rightarrow \mathit{Open}; \\
\quad \mu\, X \bullet (\mathit{open}?\mathit{acct} \rightarrow \mathit{Open} \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{deposit}?\mathit{number}?\mathit{amount} \rightarrow \mathit{Deposit} \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{balance}?\mathit{number} \rightarrow \\
\qquad\qquad\quad \mathbf{var}\ \mathit{bal} : \mathit{num} \bullet \\
\qquad\qquad\qquad \mathit{Balance};\ \mathit{out}!\mathit{bal} \rightarrow \mathit{Skip});\ X \\
\mathbf{end}
\end{array}
$$

First of all, the state is initialised, and then it is only possible to request that an account is open. Afterwards, the process recursively offers the possibility of opening an account, making a deposit, or requesting a balance.

4 Semantics: method calls

This section outlines the basis of *OhCircus*'s semantics for method calls. In Hoare and He's unifying theories, several programming paradigms are given denotational semantics in the framework of an alphabetised version of Tarski's relational calculus. Hoare and He model and establish links between imperative, functional, logical, parallel, and reactive programming. Programs are captured as predicates over states containing observations of interest; subtheories are formed by imposing healthiness conditions.

Circus is based on the combination of the imperative and reactive programming theories. The observations are the program variables and the variables tr and ref to record a CSP-style failure, $wait$ to mark a non-termination state, and $okay$ to denote divergence-freedom. For example, the deadlocked process $Stop$ is represented by the predicate below.

$$
\begin{array}{l}
tr\ \mathsf{prefix}\ tr' \wedge (\ \mathit{okay} \Rightarrow \mathit{okay}' \wedge \mathit{wait}' \wedge \\
\qquad\qquad\qquad\qquad\quad tr' = tr \wedge v' = v \wedge \\
\qquad\qquad\qquad\qquad\quad (\ \mathit{wait} \Rightarrow \mathit{ref}' = \mathit{ref}\ )\ )
\end{array}
$$

The first conjunct is a healthiness condition for every CSP process: history is never altered, or rather, the accumulated trace of events tr is unchanged. If $Stop$ is activated in a divergent state, then $okay$ will be false, and this healthiness condition is all that we can guarantee about the resulting behaviour. Otherwise, if $okay$ is true, then $Stop$ does not diverge ($okay'$ is true), does not terminate ($wait'$ is true), and does not change tr or the programming variables v. If $wait$ is true, then $Stop$'s behaviour is being considered when some sequential predecessor has not terminated; in which case, nothing changes, not even the refusal set.

In *OhCircus*, we use a class semantics based on records; methods are modelled as higher-order, predicate-valued variables, following the treatment of higher-order procedures and parameters in [19]. For example, consider the class C below, which offers an encapsulation of a natural number v with initialisation, increment, and get methods. The schema S defines the local state of C.

$$
\begin{array}{l}
\mathbf{class}\ C \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\ S\ \mathrel{\widehat=}\ [\, v : \mathbb{N}\,] \\
\quad \mathbf{initial}\ \mathit{initC}\ \mathrel{\widehat=}\ [\, S' \mid v' = 0\,] \\
\quad \mathbf{public}\ \mathit{incC}\ \mathrel{\widehat=}\ [\, \Delta S;\ i? : \mathbb{N} \mid v' = v + i?\,] \\
\quad \mathbf{public}\ \mathit{getC}\ \mathrel{\widehat=}\ [\, \Xi S;\ o! : \mathbb{N} \mid o! = v\,] \\
\mathbf{end}
\end{array}
$$

The meaning of C is a record with three labelled components (like a schema binding in Z), one for the constructor, and one for each method. These components contain program texts (unifying theories predicates) that are parametrised to make the semantics of method invocation convenient. The before and after-values of the state are abstracted as variable parameters, a new mechanism for handling variable names; inputs are abstracted as value parameters; and outputs as result parameters. We give the meaning of each of these mechanisms of parameter passing using lambda notation.
A value parameter declares a local variable that is initialised when the method is called. The semantics of a parametrised command with a value parameter is given below: a higher-order function that takes the value of the local variable as its argument and produces a program text.

$$(\,\mathbf{val}\ v : T \bullet c\,) \ \widehat{=}\ (\,\lambda\, w : T \bullet (\mathbf{var}\ v := w \bullet c)\,)$$

The value of w is the argument that is given in a method call.

A result parameter takes as argument the name of a variable with wider scope. This is the argument in a method call.

$$(\,\mathbf{res}\ v : T \bullet c\,) \ \widehat{=}\ (\,\lambda\, w : \mathsf{N} \bullet (\mathbf{var}\ v \bullet c;\ w := v)\,)$$

The parameter w is the *name* of the variable that takes the result.

The target of a method call is an extra argument, passed by value-result: $t.m(a) \mathrel{\widehat{=}} m(t, a)$. We define value-result parameter passing below. It is not directly available in *OhCircus*; it is only used to define the semantics of method calls.

$$(\,\mathbf{valres}\ v : T \bullet c\,) \ \widehat{=}$$
$$(\,\lambda\, w : \mathsf{N} \bullet (\mathbf{var}\ v := w \bullet c;\ w := v)\,)$$

The parameter of the lambda expression is again a program variable. This is an abstraction over two arguments, the before and after values, and our notation enforces this.

$$(\,\lambda\, x : \mathsf{N} \bullet c\,)(y) \ \widehat{=}\ c[y, y'/x, x']$$

In this case, the lambda calculus's rule of β-reduction is augmented in the obvious way to cope with variable parameters: elements of the syntactic category N.

If P is a predicate that gives the meaning of the body of a method of a class C, with value parameter i of type I, and result parameter o of type O, then the meaning of the method is a higher-order function P whose value is given below.

$$(\,\mathbf{valres}\ t : C;\ \mathbf{val}\ i : I;\ \mathbf{res}\ o : O \bullet P\,)$$

The parameter t represents the target of a call; its type is the class C, which denotes the set of bindings of type S, the local state of C. Multiple parameters are handled by a combination of the above definitions.

If the method is defined by a schema operation using the usual conventions of Z, the semantics of its body is the promotion of the schema from an anonymous binding to a named one. For the method $incC$ in our example, we have the following semantics.

$$\mathbf{procedure}\ incC := (\,\mathbf{valres}\ c : C;\ \mathbf{val}\ i? : \mathbb{N} \bullet$$
$$(\,\exists\, \Delta S \bullet incC \land \theta S = c \land \theta S' = c'\,)\,)$$

The parameter c, which stands for the target of a call to $incC$, is used to determine the initial value of state; the expression θC denotes a binding whose components

are those of the state S. The final value of c is the final value of the state, which is defined by $incC$.

The initialisation determines the semantics of the **new** expression: the binding defined by the initialisation schema. This should be uniquely defined. For our example, we have the following semantics.

$$\mathbf{new}\ \mathit{initC} := (\,\mu\, \mathit{initC}\,)$$

We use the Z definite description operator μ to determine the unique value defined by the initialisation.

Finally, methods that do not change the state, and have a unique result parameter, are also expressions, as long as they are deterministic. For example, the method getC is given a meaning as a function; as mentioned before, there are the usual proof obligations to show that it is indeed functional.

$$\begin{array}{l}\mathbf{function}\ \mathit{getC} : \mathbb{N} := (\,\mathbf{val}\ c : C \bullet \\ \quad (\,\mu\, o! : \mathbb{N} \bullet (\,\exists\, \Delta S \bullet \mathit{getC} \land \theta S = c\,)\,)\,)\end{array}$$

The parameter c is used to determine the initial value of the state for $getC$. The final state of $getC$ is irrelevant for the semantics of $getC$ as a function; it is existentially quantified. The definite description determines the unique output value.

As an example, suppose that d is an object of class C. We can increment d's value by 5 by applying incC to it.

$$\begin{array}{ll} & d.\mathit{incC}(5) \\ = & \{\ \text{semantics of method application}\ \} \\ & \mathit{incC}(d)(5) \\ = & \{\ \text{semantics of}\ \mathit{incC}\ \} \\ & (\,\mathbf{valres}\ c : C;\ \mathbf{val}\ i? : \mathbb{N} \bullet \\ & \quad (\,\exists\, \Delta S \bullet \mathit{incC} \land \theta S = c \land \theta S' = c'\,) \\ &)(d)(5) \\ = & \{\ \beta\text{-reduction, twice}\ \} \\ & \exists\, \Delta S \bullet \mathit{incC}[i? := 5] \land \theta S = d \land \theta S' = d' \\ = & \{\ \text{definition of}\ \mathit{incC}\ \text{and substitution of 5 for}\ i?\ \} \\ & \exists\, v, v' : \mathbb{N} \bullet v' = v + 5 \land \theta S = d \land \theta S' = d' \\ = & \{\ \text{equality of schema tuples}\ \} \\ & \exists\, v, v' : \mathbb{N} \bullet v' = v + 5 \land v = d.v \land v' = d'.v \\ = & \{\ \text{one-point rule, twice}\ \} \\ & d'.v = d.v + 5 \end{array}$$

So, the method call on d is the program text that increments d's v-component by 5.

Dynamic binding and recursion are resolved in the construction of the records that define the semantics of each of the classes. The approach follows the line of the formal semantics presented in [6] for a subset of Java.

5 Refinement

In [9], we propose a refinement strategy for *Circus* that allows iterated decompositions of processes; an overview is presented in Figure 3. Refining a *Circus* program amounts to refining its processes. A process P_1 is refined by a process P_2, if the

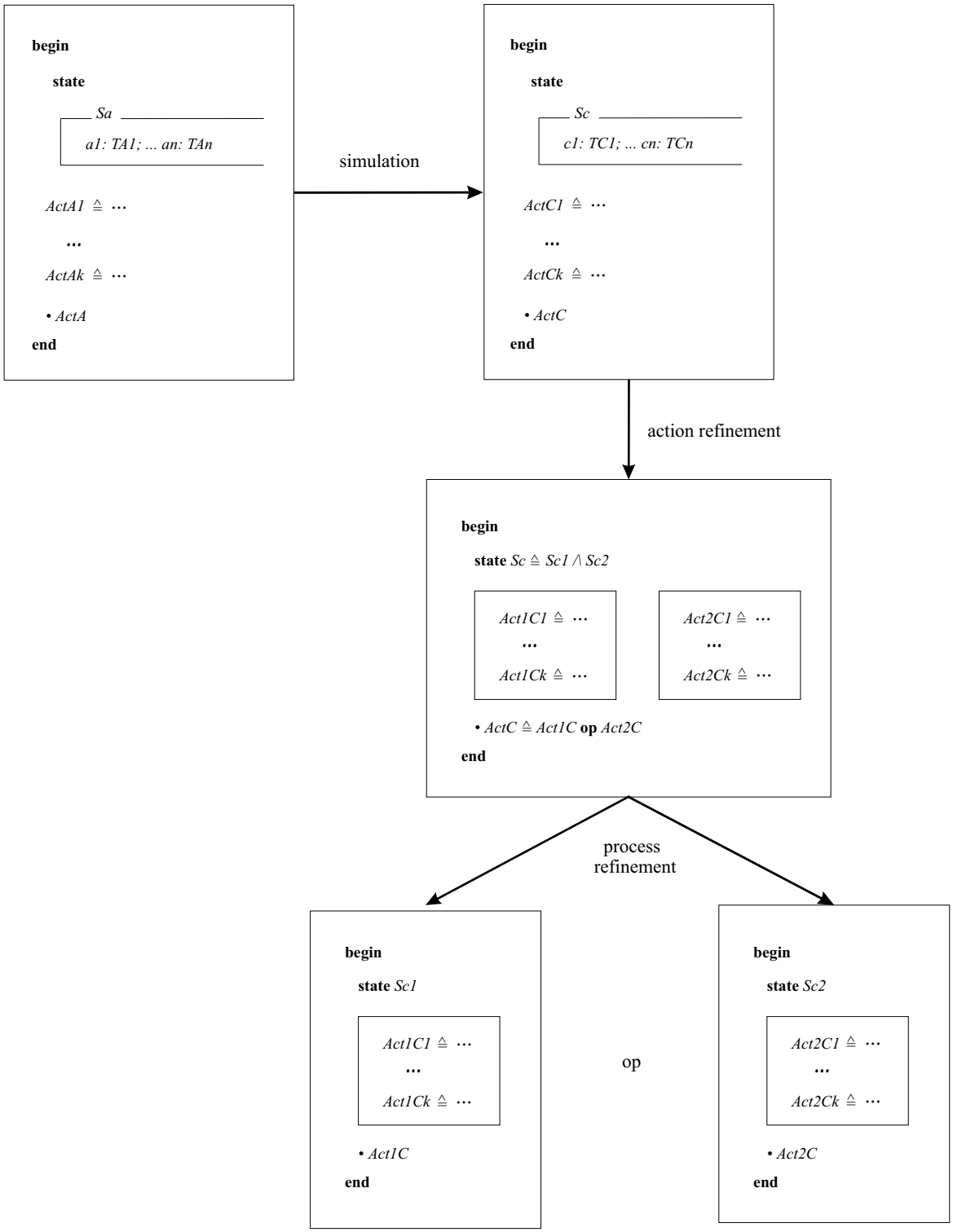

Fig. 3. Iteration of the refinement strategy

main action of P_1 is action refined by the main action of P_2, with the state components taken as local variables. Action refinement corresponds to the standard notion of refinement in Z and Dijkstra's language of guarded commands, and to failures-divergence refinement in CSP. These are both formalised in the unifying theories of programming as a single notion. Z and CSP refinement laws apply in the refinement of actions.

In order to decompose a process into, for instance, parallel sub-processes, the state and the paragraphs of the original process must be organised into two partitions: each partition is formed of a subset of the state components and a set of paragraphs that have access to them. As a result of this decomposition, each partition is promoted into a component process. The way these processes are combined is determined by the main action of the original process.

Each iteration of the strategy involves a process decomposition. In practice, however, before decomposing a process, action refinement is necessary to partition the internal structure of the process in the way explained above. Typically, if two actions (belonging to different partitions) share a state element, this shared-variable communication is transformed into a channel communication; the reason is that the processes that result from the decomposition are not allowed to share state elements (as imposed by the design of *Circus*).

Apart from action refinement, previous to a decomposition, it is often necessary to carry out some change of data representation, so that the development will progress towards concrete state components whose types are available in the target implementation platform. The relevant tools here are the laws of process simulation proposed in [9].

The purpose of this section is to illustrate, through an example, that this strategy can be conservatively extended to incorporate classes and inheritance as introduced into *OhCircus*. The extended strategy includes (possibly iterated) applications of the following steps:

- class simulation, in addition to process simulation;
- parametrised command refinement, in addition to action refinement;
- class refinement, in addition to process refinement (decomposition);
- behavioural inheritance for both classes and processes.

The concept of iteration is still the same as in the original *Circus* strategy. It is marked by an application of a process decomposition; nevertheless, this is now not confined to yield only processes: a partition may be promoted into a class.

All the other forms of refinement can freely occur inside iterations, typically as a way of partitioning the internal structure of a process for further decomposition. Particularly, class refinement can be carried out independently.

Parametrised commands are introduced as a new class paragraph in *OhCircus*; therefore action refinement is extended to deal with this new feature. Behavioural inheritance of classes and processes is inherent to the design of *OhCircus*, as discussed in Section 3.

As an example, we develop the design of a hypothetical operating system resource scheduler from an initial abstract specification. We first decompose it into a resource manager and a scheduler using standard process refinement; then, we further decompose the scheduler into a concrete scheduler and a class that encapsulates the collection of tasks to be scheduled. In the final step, we introduce priority by extending both the concrete scheduler and the task collection through behavioural inheritance.

5.1 Abstract specification of a resource scheduler

Our resource scheduler is responsible for the management of resources, and scheduling of tasks based on the availability of resources. Tasks can be modelled as a class with a single attribute: the task identifier.

$[Identifier]$

class $Task \mathrel{\widehat=}$ **begin**
  state $TState \mathrel{\widehat=} [\, id : Identifier \,]$

This identifier is initialised when the task is created and can be publicly accessed through the method *getId*.

initial *TInit*

$$TState'$$
$$id? : Identifier$$
$$\hline$$
$$id' = id?$$

public *getId*

$$\Xi TState$$
$$id! : Identifier$$
$$\hline$$
$$id! = id$$

end

Resources are abstractly represented using a given set.

$[Resource]$

The resource scheduler interacts with its environment through several channels with the following functions: the input of a new task (*in*); the output of the identifier of the executing task that has just been scheduled (*exec*); the indication that a resource became available (*available*) or is being demanded (*demand*, *demand_ok*); and the indication that the executing task has been destroyed (*out*).

channel $in : Task;\ exec : Identifier;$
$available, demand : Resource;$
$out;\ demand\_ok$

The channels *out* and *demand_ok* are used only for synchronisation; no values are communicated through them, so that their declaration does not give a type.

The way the interactions occur with the environment is captured by the process *ResourceScheduler*.

process $ResourceScheduler \mathrel{\widehat=}$ **begin**

The state components are the task that is currently executing, a set of ready tasks, a set of tasks blocked on resources, and a set of free resources.

state *RSState*

$executing : Task$
$ready : \mathbb{P}\, Task$
$block : Task \pfun Resource$
$free : \mathbb{P}\, Resource$

$ready \cap \mathrm{dom}\, block = \emptyset$
$\forall\, t_1, t_2 : ready \cup \mathrm{dom}\, block \cup \{executing\} \bullet$
$\quad t_1 \neq t_2 \Rightarrow t_1.getId() \neq t_2.getId()$
$\{\mathbf{null}, executing\} \cap \{ready \cup \mathrm{dom}\, block\} = \emptyset$
$\mathrm{ran}\, block \cap free = \emptyset$

The invariant states that the sets of ready and blocked tasks are disjoint; tasks have distinct identifers; the executing task is neither ready nor blocked; and the set of free resources is disjoint from the resources on which tasks are blocked.

As an example, we specify the insertion of a new task.

Insert

$\Delta RSState;\ t? : Task$

$t? \neq \mathbf{null}$
$\forall\, t : ready \cup \mathrm{dom}\, block \cup \{executing\} \bullet$
$\quad t \neq \mathbf{null} \Rightarrow t.getId() \neq t?.getId()$
$executing' = executing$
$ready' = ready \cup \{t?\}$
$block' = block \wedge free' = free$

The new task to be inserted must be a proper (non-null) task, and it must not be already recorded as ready or executing.

After initialisation, the main action recursively offers insertion of a new ready task (*Insert*); inclusion of a free resource, possibly releasing blocked tasks (*IncludeOrRelease*); allocation of a free resources (*Allocate*); blocking the executing task on a demanded resource which is not free (*Block*); destruction of the executing task (*Destroy*); and scheduling of a random ready task for execution, interrupting the currently executing task, if there is one (*InterruptAndExecute*). The internal choice ($\sqcap$) with *Stop* (deadlock) means that the *ResourceScheduler* decides whether or not to interrupt the currently executing task.

$\bullet$ *RSInit*;
$\mu\, X \bullet$
$\quad in?t \rightarrow Insert$
$\quad \Box\ available?r \rightarrow IncludeOrRelease$
$\quad \Box\ executing \neq \mathbf{null}\ \&$
$\quad\quad demand?r \rightarrow$
$\quad\quad\quad r \in free\ \&\ demand\_ok \rightarrow Allocate$
$\quad\quad\quad \Box\ r \notin free\ \&\ Block$
$\quad \Box\ out \rightarrow Destroy$
$\quad \Box\ ready \neq \emptyset\ \&$
$\quad\quad (\mathbf{var}\ id : Id \bullet$
$\quad\quad\quad InterruptAndExecute;\ exec!id \rightarrow Skip$
$\quad\quad) \sqcap Stop$
end

Except for *Insert*, specified previously, most actions used above are omitted since the focus of the development is on decomposition of processes, which generates

further processes and classes, and on behavioural inheritance, rather than on action refinement.

5.2 Resource scheduler internal partitioning

Aiming at decomposing the process *ResourceScheduler* into a resource manager and a scheduler, first it is necessary to transform the internal structure of the process into two partitions. These will then be promoted into the relevant processes, as explained early in this section.

The sharing of state between the partitions is replaced by explicit channel communications. For instance, the action *IncludeOrRelease* is described as a cooperation of the partitions. When the resource manager partition receives from the environment a resource through the channel *available*, it recovers the tasks blocked on that resource and sends them through a channel *unblock* to the scheduler partition, which adds them to the set of ready tasks.

channel $unblock : \mathbb{P}\, Task$

Further, when the scheduler partition receives a demand for a resource, it interacts with the resource manager partition, communicating the currently executing task and the resource itself, using the new channel *request*; as a result, the resource manager partition communicates back informing whether the resource is free (*request_ok*) or not (*block*).

channel $request : Task \times Resource;$
$request\_ok;\ block$
chanset $RSProtocol \mathrel{\widehat=}$
$\{\!|\, block, unblock, request, request\_ok \,|\!\}$

The identifier *RSProtocol* is introduced to name the above set of channels; it is an abbreviation for future convenience.

The interactions between the two partitions are made precise in the refined version of the *ResourceScheduler* presented below.

process $ResourceScheduler \mathrel{\widehat=}$ **begin**

The state of the process *ResourceScheduler* is split into two disjoint schemas, so that the original state is now expressed as the conjunction of these two schemas. The first schema will become the state of the scheduler and the second one the state of the resource manager.

SState

$executing : Task;\ ready : \mathbb{P}\, Task$

$executing \notin ready$

RMState

$block : Task \pfun Resource;\ free : \mathbb{P}\, Resource$

$\mathrm{ran}\, block \cap free = \emptyset$

state $RSState \mathrel{\widehat=} SState \land RMState$

Since the state is partitioned, the part of the invariant which relates elements of the two states cannot be explicitly stated anymore. Of course, as the partitioning is a

data refinement, the invariant still holds implicitly; the relevant proof obligations amount to proving that all the operations preserve the invariant. They are omitted for conciseness.

The initialisation is partitioned in a similar way. Concerning the other actions (whose behaviour has been briefly described in the previous subsection), *Insert*, *Destroy* and *ExecuteAndInterrupt* involve solely scheduling activities; therefore, they are included in the scheduler partition. On the other hand, *Allocate* has to do only with resource managing, and is part of the associated partition.

The actions *Block* and *IncludeOrRelease* refer to components of both partitions. The action *Block* gives rise to two actions: *SBlock*, in the scheduler partition, which assigns **null** to *executing*, and *RMBlock*, in the resource manager partition, which blocks the executing task on an unavailable resource. The action *IncludeOrRelease* is replaced with three actions: *Include* and *Release*, in the resource manager partition, which include a new resource and release tasks blocked on a resource that becomes free, and *Unblock*, in the scheduler partition, which includes the liberated tasks in the set of free tasks.

The main actions of the two partitions formalise each allowed external behaviour, as well as the interactions between the partitions. They are given as separate actions below.

$$
\begin{array}{l}
SAction \mathrel{\widehat=} \\
\quad SInit; \\
\quad \mu\, X \bullet \\
\quad\quad (\quad in?t \rightarrow Insert \\
\quad\quad\ \Box\ executing \neq \mathbf{null}\ \& \\
\quad\quad\quad\quad demand?r \rightarrow request!executing!r \rightarrow \\
\quad\quad\quad\quad\quad request\_ok \rightarrow demand\_ok \rightarrow Skip \\
\quad\quad\quad\quad\quad \Box \\
\quad\quad\quad\quad\quad block \rightarrow Block \\
\quad\quad\quad \Box\ out \rightarrow Destroy \\
\quad\quad\ \Box\ executing \neq \mathbf{null} \land ready \neq \emptyset\ \& \\
\quad\quad\quad\quad (\mathbf{var}\ id : Identifier \bullet \\
\quad\quad\quad\quad\quad InterruptAndExecute;\ exec!id \rightarrow Skip \\
\quad\quad\quad\quad) \sqcap Stop \\
\quad\quad\ \Box\ unblock?ts \rightarrow Unblock);\ X
\end{array}
$$

This is very similar to the main action of *ResourceScheduler*, except that it interacts with the action below.

$$
\begin{array}{l}
RMAction \mathrel{\widehat=} \\
\quad RMInit; \\
\quad \mu\, X \bullet \\
\quad\quad (\quad available?r \rightarrow \\
\quad\quad\quad\quad\quad r \notin \operatorname{ran} block\ \&\ Include \\
\quad\quad\quad\quad \Box\ r \in \operatorname{ran} block\ \& \\
\quad\quad\quad\quad\quad\quad \mathbf{var}\ ts : \mathbb{P}\ Task \bullet \\
\quad\quad\quad\quad\quad\quad\quad Release;\ unblock!ts \rightarrow Skip \\
\quad\quad\ \Box\ request?t?r\ \& \\
\quad\quad\quad\quad\quad r \in free\ \&\ Allocate;\ request\_ok \rightarrow Skip \\
\quad\quad\quad\quad \Box\ r \notin free\ \&\ block \rightarrow Block);\ X
\end{array}
$$

The behaviour of the refined *ResourceScheduler* process can then be given in terms of the parallel composition of the above two actions, hiding the new channels intro-

duced for synchronisation between the two partitions.

• ($SAction$
  $[\![\alpha SState \mid RSProtocol \mid \alpha RMState]\!]$
  $RMAction$) $\setminus$ $RSProtocol$
end

The syntax of the parallel operator makes explicit that the *SAction* can modify only components of *SState*, whereas *RMAction* is allowed to change only elements of *RMState*. As already mentioned, this extra information is not needed in the parallel composition of processes, whose states are encapsulated, but is fundamental in the combination of actions to avoid conflicts in the modification of variables.

The refinement carried out in this subsection is a pure (although elaborate) action refinement. Formally, it can be justified using the notions and laws presented in [9], adapted to handle the fact that *Task* is a class type. This detailed justification is out of the scope of this paper.

5.3 Decomposition: resource manager and scheduler processes

Once a process is partitioned, like *ResourceScheduler* was in the previous subsection, its partitions can be promoted into processes, as formalised in [9]. Consider that *Scheduler* is a process with state *SState*, actions *SInit*, *Insert*, *Destroy*, *ExecuteAndInterrupt*, *SBlock* and *Unblock*, and main action *SAction*. Assume also that *ResourceManager* is a process with state *RMState*, actions *Allocate*, *Include*, *Release* and *RMBlock*, and main action *RMAction*. In this case, the process *ResourceScheduler* can be redefined as follows.

process $ResourceScheduler \mathrel{\widehat=}$
  ($Scheduler$
    $[\![\alpha SState \mid RSProtocol \mid \alpha RMState]\!]$
  $ResourceManager$) $\setminus$ $RSProtocol$

This marks the end of the first iteration of our development.

5.4 Scheduler decomposition: concrete scheduler and task collection

So far we have illustrated refinement steps that can be justified using the strategy developed for *Circus* [9]. The remaining steps of our development focus on a more concrete design of *Scheduler*, and single out the features introduced in *OhCircus*: in particular, classes and inheritance.

In this refinement step the *Scheduler* process is decomposed into a more concrete version of the scheduler and a class that encapsulates the collection of tasks to be scheduled. Analogously to the previous decomposition, before the actual splitting of the process, the relevant partitions need to be identified and the internal structure of the process needs to be modified, showing explicitly how these partitions cooperate to preserve the original behavior.

In the case of decomposing a process into two other processes, the two partitions are identified simultaneously, and their cooperation is formalised by a process algebra operator (like parallelism, as illustrated in the previous step) used to combine the corresponding main actions.

Concerning a process decomposition that generates a process and a class, the partition that will be promoted into a class is identified first, and the class declaration is actually introduced. The original process is then data refined to become a client of the generated class. Therefore the cooperation between the resulting class

class *TaskCollection* $\widehat{=}$ **begin**

state *TCState*

$tasks : \mathbb{P}\ Task$

$\forall\, t : tasks \bullet t \neq \mathbf{null}$

$\forall\, t_1, t_2 : tasks \mid t_1 \neq t_2 \bullet t_1.getId() \neq t_2.getId()$

initial $TCInit \mathrel{\widehat{=}} [\, TCState' \mid tasks' = \emptyset \,]$

public *insert*

$\Delta TCState$

$t? : Task$

$\forall\, t : tasks \bullet t.getId() \neq t?.getId()$

$tasks' = tasks \cup \{t?\}$

public *remove* $\widehat{=} [\, \Delta TCState;\ t! : Task \mid t! \in tasks \land tasks' = tasks \setminus \{t!\} \,]$

public *getTasks* $\widehat{=} [\, \Xi TCState;\ tasks! : \mathbb{P}\ Task \mid tasks! = tasks \,]$

end

Fig. 4. Class *TaskCollection* specification.

and process is established by the clientship relation itself (contrasting with a process algebra operator, as in the previous case). The idea for decomposing a process into a process and a class is similar to the well-known refactoring *extracting class* in the object-oriented paradigm [17].

In our example, the partition that gives rise to a new class is that formed of the state component *ready* (of *SState*) and the several operations over this component that appear in the actions of *Scheduler*. Extracting this partition results in the class in Figure 4, with a single attribute (a set of tasks) and methods for including and removing tasks, and for accessing the entire set of tasks. Tasks are randomly removed; this is a consequence of the fact that scheduling is random in the process *Scheduler*, which is the source of this decomposition.

The other product of this decomposition is the *Scheduler* itself, adjusted to be a client of the above class. A simple data refinement justifies this transformation.

This concludes the second iteration of our case study.

5.5 Introducing priority tasks

The target scheduler of our development is one which allocates tasks based on priority. The purpose of this step is to introduce the class *PriorityTask*, which inherits from *Task* and includes a new attribute to record priority.

We also introduce *PriorityTaskCollection* as a subclass of *TaskCollection*. In principle, this should not be necessary, since *TaskCollection* can store instances of *PriorityTask*. Nevertheless, the inherited collection is introduced because it redefines the *remove* method responsible for selecting the next element for scheduling: while in the original collection this choice is totally arbitrary, in the inherited one, the task with higher priority is returned.

First we introduce constants to fix the maximum priority and define the priority range.

$maxPriority : \mathbb{N}_1$

$Priority == 1 \,..\, maxPriority$

In the subclass *PriorityTask*, presented in Figure 5, we extend the class *Task* with a new attribute to record priority. Apart from the constructor, set and get methods are introduced for the new attribute.

The introduction of a new class is a simple refinement step; however, since *PriorityTask* is declared as a subclass of *Task*, a proof obligation to ensure behavioural subclassing is generated. The preservation of behaviour is intuitive, since it involves no method redefinition and the extra methods refer only to the new attribute.

Analogously, we extend *TaskCollection* with priorities. Although the intention is to store in the new collection only instances of *PriorityTask*, it would not be valid to enforce this requirement, say as an invariant in the class definition. The reason is that the original collection allows both the inclusion and deletion of ordinary tasks. In order to ensure behavioural subclassing, such a behaviour cannot be forbidden.

class *PriorityTaskCollection* $\widehat{=}$
extends *TaskCollection* **begin**

A new attribute is introduced to store the tasks with priority. The invariant states that this set of priority tasks is a subset of the original set.

state *PTCState*

$$priTasks : \mathbb{P}\, PriorityTask$$

$$priTasks = \{\, t : tasks \mid t \,\mathbf{instanceof}\, PriorityTask \,\}$$

The *remove* method is redefined to choose one of the tasks with highest priority (1 is considered a higher priority than 2, and so on); this is used as a new scheduling policy in the scheduler designed in the next section.

public *remove*

$$\Delta PTCState$$
$$t! : Task$$

$$t! \in task$$
$$tasks' = tasks \setminus \{\, t! \,\}$$
$$priTasks \neq \emptyset \Rightarrow$$
$$\quad t! \in priTasks \land$$
$$\quad \forall\, tp : priTasks \bullet$$
$$\qquad t!.getPriority() \leq tp.getPriority()$$

When *priTasks* is empty, and *tasks* is not, we have a subtle situation. In this case, an arbitrary (non-priority) task is chosen, so the redefined method works just as the original *remove* method, as should be expected.

The method *insert* is also redefined.

public *insert*

$$\Delta PTCState$$
$$t? : Task$$

$$\forall\, t : tasks \bullet t.getId() \neq t?.getId()$$
$$tasks' = tasks \cup \{t?\}$$

end

class *PriorityTask* $\widehat{=}$ **extends** *Task* **begin**

state *PTState* $\widehat{=}$ [*priority* : *Priority*]

initial *PTInit* $\widehat{=}$ **val** *id* : *Identifier*; *initPri* : *Priority* $\bullet$
TInit; *priority* := *initPri*

public *setPriority*

$\Delta PTState$
$\Xi TState$
priority? : *Priority*

---

priority$'$ = *priority*?

public *getPriority* $\widehat{=}$ [$\Xi TState$; *priority*! : *Priority* | *priority*! = *priority*]

end

Fig. 5. Class *PriorityTask* specification.

The only modification to its original definition is that the redefined version acts on the extended state (*PTCState*). If the input task $t?$ happens to be a priority task, it is included both in *tasks* (explicitly) and in *priTasks* (due to the invariant).

Concerning preservation of behaviour, only *remove* and *insert* are redefined. The redefined version of *insert*, as already explained, updates the component *task* exactly as before, and eventually also updates the new attribute *priTasks*. Considering the redefinition of *remove*, it still removes a task from the set *tasks*, but eventually one of those with highest priority, in which case it also updates the new attribute *priTasks*. Therefore, regarding the original attribute, the redefined version clearly strengthens the postcondition. This can be discharged by simple predicate calculation.

5.6 A priority scheduler

A priority scheduler can be designed as a specialisation of the process *Scheduler*, as *OhCircus* allows inheritance of processes, in addition to class inheritance. In our example, no action needs to be added, only the invariant is strengthened, as presented below. If there are no extra components, we can define the state just as a predicate introduced using $\vdash$. It is conjoined to the invariant of the state of the superprocess (or superclass).

process *PriorityScheduler* $\widehat{=}$ **extends** *Scheduler*
begin
  state $\vdash$ *tasks* **instanceof** *PriorityTaskCollection*
end

As tasks can store both ordinary and priority tasks, the executing task might eventually be a non-priority one. This is why we do not enforce that the *executing* attribute of the *PriorityScheduler* be an instance of *PriorityTask*.

As the main action is missing, it is assumed to be *Skip*. It is put in parallel with that in the superprocess. In this case, the process inheritance is very simple, and it is intuitive that the behaviour of the original process is preserved.

6 Conclusions

This paper has presented *OhCircus*, a language for specification and refinement, which integrates Z, CSP, refinement calculi constructs, and object-oriented concepts. We have discussed our approach to its model; it unifies elaborate mechanisms for defining data (classes) and control behaviour (processes). Using the unifying theories of programming, it is possible to combine these constructs and still specify and reason in the resulting formalism without deviating from all we have learned working with each notation in isolation.

By describing our work, in particular on *OhCircus*, and in general on *Circus*, this paper makes a contribution to the debate on states *versus* events. Many specification methods are based either on states or on events; they rely on the fact that these approaches are equivalent, since an event can be modelled as a state change, and a state can be modelled as an equivalence class of sequences of actions. These methods have taken very different formal directions, and they tend to differ in practice.

In fact, neither approach is dominant in *Circus*. We followed two important principles in designing *Circus*: states and events should be semantically integrated; and system architecture should be uncommitted. Although *Circus* combines Z and CSP, it is not part of the tradition that views this combination as requiring a notion of communicating abstract data-types. Instead, state transitions and events are decoupled, each occurring when they need to, according to the behaviour required. This allows *Circus* to encompass a wider variety of programming styles and paradigms.

In the *Circus* refinement strategy, the starting point is an abstract, and usually centralised specification that is progressively transformed into a concrete and distributed architecture, based on laws for process decomposition and (data) refinement. During this process, explicit global invariants become implicit distributed invariants. State-based descriptions become reflected in distributed reactive behaviour. The global invariant actually guides the discovery of the reactive design.

Within the context of *OhCircus*, this strategy is extended to handle the introduction of classes, as a means of further structuring the states of processes. New definitions and laws are needed to support the extended strategy, but the overall approach is still valid, as illustrated by the development in the previous section.

Circus and *OhCircus* include novel specification devices that play a central rôle in the refinement strategy. For example, the state in a *Circus* program is carefully partitioned between processes, which leads to a clean programming model free from race conditions. For actions, the partitioning of the state is imposed by the operators for parallelism.

If we want to introduce parallelism by splitting a process into sub-processes, a natural way to proceed is by stepwise development. Since we allow shared variables inside a process, we can separate the task of dealing with the event structure (concurrency and communication) from that of dealing with the state structure (partitioning the variables). During development, we introduce actions that share state. Next, we shift state components into particular subsets, replacing references to shared state by event-based communication. This continues until the process structure emerges. In this way, we use the interplay between states and events to great advantage during development.

Something similar happens in the treatment of objects in *OhCircus*: visibility is a code-level constraint. During development we can refer freely to private and protected attributes of a class as we construct and refine its clients. The final code, however, can only rely on method calls to access and update such attributes.

At the level of actions, assertional reasoning is possible since assertions are special forms of specification statements. In particular, assertions over state properties can be used to restrict reactive behaviour. This is possible because *Circus* allows the free combination of state operations and events. As such, an extended form of

Hoare logic may be adopted as a reasoning technique, which leads, in the usual way, to predicate transformers and to refinement calculi.

In [3], a framework called St.Eve is presented with the aim of supporting the classification of specification languages and techniques with respect to their approach to the definition of state and behavioural aspects of a system. In the St.Eve terminology, an *OhCircus* event is a communication, just as in CSP. A state is a binding, a mapping of values to state components, just as in Z. A specification is a sequence of paragraphs, again as in Z; we have, however, new forms of paragraphs for channel, process, and class definitions. The semantics is a predicative rendering of alphabetised relations.

More interesting, *OhCircus* supports three forms of system decomposition policies or constraints: actions, processes, and classes. Inside a process, actions support descriptions with shared events and variables; their generality is controlled by the structure of processes and the parallelism operators. In *Circus* and *OhCircus*, the parallel composition operators for actions require that each of the parallel actions have update access restricted to a partition of the state; all actions have read access to the before state, but can change only variables in their own partition. Processes are based on pure event constraints, since their states are encapsulated. Finally, classes support a pure state partitioning.

We intend to characterise behavioural refinement for processes and classes in the semantic framework of *OhCircus*, independently of the obvious formulation in terms of forwards simulation. Maybe we will be able to provide simple proof obligations related to subprocess declarations that do not require that the actions of the superprocess and the subprocess are executed in parallel. More flexibility may be of practical use.

As *OhCircus* has most of the central design elements of notations like Real Time UML, a refinement strategy for *OhCircus* is illuminating in the formalisation of industrial development practices, possibly through a mapping between these languages. An initial result has been presented in [28].

We have already shown that the calculational style of reasoning is possible for *Circus*. In [7, 5], we considered refinement of object-oriented programs. We plan to bring these results together in a unified framework of programming. The unified language and model presented in this paper is the basis for all these further research investigations.

Acknowledgements

The authors benefitted from discussions with participants of the St.Eve workshop, that took place as an FME'2003 satellite event in Pisa. We are also grateful for Graeme Smith's comments on a previous draft of this paper. This work is partially funded by QinetiQ and the Royal Society.

References

1. J-R. Abrial. *The B-Book: Assigning Progams to Meanings*. Cambridge University Press, 1996.
2. R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo, Finland, 1987. Ser. A No. 55.
3. T. Bolognesi. On State-oriented versus Object-oriented Thinking in Formal Behavioural Specifications. Technical Report 2003-TR-20, ISTI – Istituto di Scienza e Tecnologie della Informazione Alessandro Faedo, 2003.
4. P. H. M. Borba, A. C. A. Sampaio, A. L. C. Cavalcanti, and M. L. Cornélio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100, 2004.

5. P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A Refinement Algebra for Object-oriented Programming. In *European Conference on Object-oriented Programming 2003 — ECOOP 2003*, volume 2743 of *Lecture Notes in Computer Science*, pages 457–482. Springer-Verlag, 2003.
6. A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713 – 728, 2000.
7. A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods — Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 471 – 490. Springer-Verlag, 2002.
8. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of Actions in *Circus*. In J. Derrick, E. Boiten, J. C. P. Woodcock, and J. Wright, editors, *Proceedings of REFINE'2002*, volume 70 of *Eletronic Notes in Theoretical Computer Science*. Elsevier, 2002.
9. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
10. J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.
11. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
12. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume 2, pages 423 – 438. Chapman & Hall, 1997.
13. C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*. Springer-Verlag, 1998.
14. C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.
15. C. Fisher and H. Wehrheim. Behavioural Subtyping Relations for Object-oriented Formalisms. In T. Rus, editor, *AMAST 2000: International Conference on Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, 2000.
16. Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 2.28*, 1999.
17. M. Fowler. *Refactoring*. Addison-Wesley, 1999.
18. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
19. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
20. J. Ichbiah. Rationale for the Design of the Ada Programming Language. *ACM SIGPLAN Notices*, 14(6B (special issue)), 1979.
21. G. Jones. *Programming in* occam 2. Prentice-Hall, 1988.
22. B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150 – 177, 2000.
23. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
24. J. M. Morris and A. Bunkenburg. Partiality and Nondeterminacy in Program Proofs. *Formal Aspects of Computer Science*, 10:76 – 96, 1998.
25. A. C. Mota and A. C. A. Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, 40:59 – 96, 2001.
26. M. V. M. Oliveira and A. L. C. Cavalcanti. From *Circus* to JCSP. In J. Davies et al., editor, *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 320 – 340. Springer-Verlag, November 2004.
27. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
28. A. C. A. Sampaio, A. C. Mota, and R. T. Ramos. Class and Capsule Refinement in UML for Real Time. In A. L. C. Cavalcanti and P. D. L. Machado, editors, *WMF 2003: 6th Brazilian Workshop on Formal Methods*, volume 95 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003. Invited paper.
29. A. C. A. Sampaio, A. C. Mota, and R. T. Ramos. Class and Capsule Refinement in UML for Real Time . In A. L. C. Cavalcanti and P. Machado, editors, *WMF 2003: 6th*

Brazilian Workshop on Formal Methods, volume 95 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

30. A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451 – 470. Springer-Verlag, 2002.
31. S. Schneider and H. Treharne. Communicating B Machines. In D. Bert, J. Bowen, M. Henson, and K. Robinson, editors, *ZB'2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 416 – 435, 2002.
32. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 1999.
33. C. Sweeney and M. Bowen. *D-1-C Handel-C Language Reference Manual*. Embedded Solutions Ltd, 1998.
34. H. Wehrheim. Subtyping patterns for active objects. In *8ter Workshop des GI Arbeitskreises GROOM (Grundlagen objekt-orientierter Modellierung)*, 2000.
35. P. H. Welch, J. R. Aldous, and J. Foster. CSP Networking for Java (JCSP.net). In *Global and Collaborative Computing Workshop Proceedings, ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695 – 708. Springer-Verlag, 2002.
36. J. C. P. Woodcock and A. L. C. Cavalcanti. A Concurrent Language for Refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.
37. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184 – 203. Springer-Verlag, 2002.
38. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.