Algebraic Laws for Process Subtyping

José Dihego[1,2], Pedro Antonino[1], and Augusto Sampaio[1]

[1] Centro de Informática, Universidade Federal de Pernambuco, Recife-PE, Brazil
{jdso,prga2,acas}@cin.ufpe.br
[2] IFBA, Feira de Santana-BA, Brazil
{jose.dihego}@ifba.edu.br

Abstract. This work presents a conservative extension of *OhCircus*, a concurrent specification language, which integrates CSP, Z, object-orientation and embeds a refinement calculus. This extension supports the definition of process inheritance, where control flow, operations and state components are eligible for reuse. We present the extended *OhCircus* grammar and, based on Hoare and He's Unifying Theories of Programming, we give the formal semantics of process inheritance and its supporting constructs. The main contribution of this work is a set of sound algebraic laws for process inheritance. The proposed laws are exercised in the development of a case study.

Keywords: Behavioural Subtyping, *OhCircus*, UTP, Algebraic Laws.

1 Introduction

Several formalisms offer support for modelling behavioural and data aspects of a system. For instance, CSP-OZ [9], CSP-B [19], Mosca (VDM+CCS) [21] and *Circus* [15] are some contributions in this direction. Particularly, *Circus* is a combination of Z [20] and CSP [10], which includes constructions in the style of Morgan's refinement calculus [13]. With the intention to also handle object orientation, the *OhCircus* [6] language has been proposed as a conservative extension of *Circus*.

Circus has a refinement calculus that embodies a comprehensive set of laws [5,15,18]. These laws are also valid for *OhCircus*. Nevertheless, although there is a notion of process inheritance in *OhCircus*, the current calculus does not include any laws for dealing with process inheritance. The laws developed in Section 4 aim to contribute to a more comprehensive set of algebraic laws for *OhCircus*, taking into account this relevant language feature.

Class inheritance, in the object-orientated paradigm, is a well-established concept [12]; several works, based on the substitutability principle, have developed theories that recognize suitable inheritance notions between classes [1,12]. On the other hand, the semantics of process inheritance is not consolidated. Some of the most well known works [9,14,22] have used the failures behavioural model of CSP to define a process inheritance relation.

Process inheritance, as originally defined for *OhCircus*, has a practical disadvantage: there is no way of explicitly referencing the inherited elements in the subprocesses; as a consequence, there is no support for taking advantage of redefinitions, which are strongly connected with the concept of inheritance. As our first contribution, we develop an extended syntax for *OhCircus*, which allows reuse of all the process elements, but still keeping processes as encapsulated units concerning their use in process compositions. Typing rules are developed to validate programs considering the new syntax, and a formal semantics is given in the Unifying Theories of Programming (UTP) [11]. The second major contribution of this work is the proposal of sound laws to support the stepwise introduction or elimination of process inheritance and process elements in the presence of this feature. We have also mechanised these rules based on the Eclipse Modelling Framework and on the Xtext and the ATL integrated tools. The overall approach is illustrated through the development of a case study.

In the next section we briefly introduce *OhCircus* through an example, already considering the extended grammar we propose. The semantics for process inheritance is presented in Section 3. A selection of the proposed laws is given in Section 4; the laws are exercised in a case study in Section 5. Finally, in Section 6, we present our conclusions and future work.

2 Process Inheritance with Code Reuse

We have extended the syntax of *OhCircus* in two central ways: the creation of a new access level to allow visibility of process elements (state and schema operation) by subprocesses (like the protected mechanism in Java) and the addition of a new clause to define Z schemas [20], very similar to the Z schema inclusion feature, with the aim of allowing schema redefinitions.

As originally designed, a process, both in *Circus* and *OhCircus*, is a black box with interaction points through channels that exhibit a behaviour defined by its main action. Actually, in a subprocess specification, all the definitions of the superprocess (state components, actions, and auxiliary definitions) are in scope; this has been motivated by the fact that the main action of the subprocess is implicitly composed in parallel with the main action of the superprocess. On the other hand, there is no notation for explicitly referencing the inherited elements for supporting code reuse, for instance, in operation redefinitions. The effort of introducing inheritance with this process structure is prohibitive because the benefits of code reuse cannot be reached and the introduction of a type hierarchy, by itself, is not enough to justify inheritance, from a practical perspective.

The syntax for the proposed extensions is presented in Figure 1, where the three central elements of our strategy are underlined. A process is a sequence of paragraphs, possibly including a state defined in the form of a Z schema (formed of variable declarations and a predicate), followed by a main action that captures the active behaviour of the process. A process paragraph (PParagraph) includes Z schemas (typically defining operations) and auxiliary actions used by the main action; a paragraph is allowed to refer to one or more Z schemas defined in the process itself or inherited from its superprocesses, in any level of inheritance.

$$
\begin{array}{lll}
\mathsf{OhProcessDefinition} & ::= & \textbf{process}\ \mathsf{N} \mathrel{\widehat=} [\textbf{extends}\ \mathsf{N}]\ \mathsf{Process} \\
\mathsf{Process} & ::= & \textbf{begin} \\
 & & \quad \mathsf{PParagraph}^{*} \\
 & & \quad [\textbf{state}\ \mathsf{N}\ \mathsf{Schema\text{-}Exp} \mid \mathsf{Constraint}] \\
 & & \quad \mathsf{PParagraph}^{*} \\
 & & \quad \bullet\ \mathsf{Action} \\
 & & \textbf{end} \\
 & \mid & \ldots \\
\mathsf{PParagraph} & ::= & \mathsf{SchemaText} \mid \mathsf{N} \mathrel{\widehat=} \mathsf{Action} \\
 & \mid & \underline{[\mathsf{PQualifier}]}\ \mathsf{N}\ \mathsf{SchemaText} \\
\mathsf{SchemaText} & ::= & ((\Xi \mid \Delta)\ \mathsf{N})^{+}\ [\mathsf{Declaration}^{+}]\ \underline{[\textbf{super}.\mathsf{N}^{+}]}\ [\mathsf{Predicate}] \\
\mathsf{Schema\text{-}Exp} & ::= & (\underline{[\mathsf{PQualifier}]}\ \mathsf{Declaration})^{*} \\
\mathsf{PQualifier} & ::= & \underline{\textbf{protected}} \\
\mathsf{N} & ::= & \textit{identifier}
\end{array}
$$

Fig. 1. *OhCircus* extended syntax

A process might extend only one process; multiple inheritance is not allowed, mainly due to the possible duplication and ambiguity that arise from this feature.

A Z schema can be defined using an explicitly access modifier, **protected**, or, if no modifier is used, the default level (inherited but not directly referenced by subprocesses) is adopted. Only Z schemas in the protected level are eligible for use in a **super** clause. The overriding of protected schemas is also supported and it allows a subprocess to redefine a protected schema introduced in or inherited by the closest superprocess up in the inheritance tree.

Similarly to schemas it is allowed to define an access level for each state component. It generates some restrictions in the subprocess state component declaration. This new syntax and its restrictions are exemplified in the sequel.

2.1 An Example

We model the standard concept of an abstract unbounded buffer considering the extensions we propose to *OhCircus* (see Figure 2). The relevant channels are *start*, *input* and *output*. The first one is a signal for the buffer initialization, and the other two communicate inputs and outputs, respectively. We introduce the process *Buffer* that implements the buffer concept in *OhCircus*. The singleton state component of the *Buffer* process is a sequence of natural numbers, which is used to implement the behaviour of a queue. It is initialized by the *Init* schema.

The behaviour of the buffer is to input and output on different channels, according to a FIFO policy. Whenever it is empty, it cannot refuse to input and, whenever it is non-empty, it cannot refuse to output. The schema *Add* receives and adds an element to the buffer, by storing it in the end of the sequence representing the queue. The schema *Remove* retrieves and removes an element from the buffer (the head of the sequence). The behaviour of the *Buffer* process is given by a main action in the style of CSP, but may also reference the process

channel $input, output : \mathbb{N}$
channel $start$

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

state St
protected $queue : \operatorname{seq} \mathbb{N}$

$Init$
St'
$queue' = \langle\rangle$

protected Add
ΔSt
$e? : \mathbb{N}$
$queue' = \langle e? \rangle \frown queue$

protected $Remove$
ΔSt
$e! : \mathbb{N}$
$\#queue > 0$
$e! = last\ queue$
$queue' = front\ queue$

$Input \mathrel{\widehat=} input?e \rightarrow Add$
$Output \mathrel{\widehat=} (\#queue > 0) \&$
  var $e : \mathbb{N} \bullet Remove;\ output!e \rightarrow SKIP$

$\bullet\ start \rightarrow Init;$
  $\mu X \bullet (Input \mathbin{\Box} Output);\ X$
end

Fig. 2. *Buffer* specification in (extended) *OhCircus*

paragraphs. The process *Buffer*, after engaging in an event communicated by the *start* channel, executes its initializer *Init*. The operator ';' stands for sequential composition, and indicates that if and only if $start \rightarrow Init$ finishes successfully the process behaves like $\mu X \bullet (A);\ X$, a recursive process that behaves like A and if A terminates successfully it behaves again like A, and so on. In our example, A stands for an external choice of input and output actions ($Input \mathbin{\Box} Output$). The *Input* action receives an input value through the channel *input* and then behaves like the *Add* operation; this establishes a binding between the variable e in the input communication and the homonymous input variable in the schema *Add*. In the case of the *Output* action, a local variable is introduced to create a binding with the corresponding variable in the *Remove* schema. Then its value is communicated through the *output* channel.

We provide specialisation of this abstract unbounded buffer, *BufferImp* (see Figure 3). It has a flexible capacity that duplicates whenever it is full. It is possible to query the ratio size/capacity. Furthermore, it provides double addition capability.

The schema *Add* in *BufferImp* uses the **super** clause to reuse the original behaviour of the *Add* operation of *Buffer*, plus duplicating the buffer length whenever it is full. The schema *Add*2 adds two elements to the buffer by sequential executions of the *Add* operation. The operation *FactorCapacity* gives the ratio between the buffer's size and length. In Z, Ξ is used to indicate that the state is unchanged by the operation, whereas Δ indicates the possibility of state modification. The main action, after initializing the buffer initial length, recursively offers the behaviour $Input2 \mathbin{\Box} Fac$. The local action *Input*2 receives two elements through the channel *input*2, adding them to the buffer by behaving as *Add*2. The action *Fac* uses the Z schema *FactorCapacity* to inform the ratio size/length.

The semantics of process inheritance is given by the parallel composition of the main action of the subprocess with that of its immediate superprocess.

channel $\mathit{length} : \mathbb{N}_1$
channel $\mathit{input2} : \mathbb{N}.\mathbb{N}$
channel $\mathit{fc} : \mathbb{R}$

process $\mathit{BufferImp} \mathrel{\widehat=}$
extends Buffer **begin**

state St
$\mathit{length} : \mathbb{N}_1$
$\#\mathit{queue} \leq \mathit{length}$

Init
St'
$\mathit{length}? : \mathbb{N}_1$
$\mathit{length}' = \mathit{length}?$

Add
$\Delta \mathit{St}$
super Add
$\#\mathit{queue} = \mathit{length} \Rightarrow \mathit{length}' = \mathit{length} * 2$

$\mathit{Add2} \mathrel{\widehat=} \mathit{Add} \,⨾\, \mathit{Add}\left[f?/e?\right]$

$\mathit{FactorCapacity}$
$\Xi \mathit{St}$
$\mathit{fc}! : \mathbb{R}$
$\mathit{fc}! = \#\mathit{queue}\ \mathsf{div}\ \mathit{length}$

$\mathit{Input2} \mathrel{\widehat=} \mathit{input2}?e?f \rightarrow \mathit{Add2}$
$\mathit{Fac} \mathrel{\widehat=}$ **var** $\mathit{fc} : \mathbb{N} \bullet \mathit{FactorCapacity};\ \mathit{fc}!\mathit{fc} \rightarrow \mathit{SKIP}$

$\bullet\ \mathit{length}?\mathit{length} \rightarrow \mathit{Init};$
$\mu X \bullet (\mathit{Input2} \mathbin{\square} \mathit{Fac});\ X$
end

Fig. 3. *BufferImp*: a subprocess of *Buffer*

The formal details are the subject of the next section. In our example, the semantics of *BufferImp* is given by the parallel composition of its main action with the *Buffer* main action. The schema *Add* is redefined in *BufferImp* and, by dynamic binding, the redefined version is the one considered when the main action of *BufferImp* is executed. Although relatively simple, this example already illustrates one of our contributions: the extension of *OhCircus* to allow operation redefinition and reuse in process inheritance.

3 Semantics

Three models to define the behaviour of a CSP process are formally established in [10,17]: traces, failures and failures-divergences. A trace $s \in \mathit{traces}(P)$ of a process P is a finite sequence of symbols recording the events in which it has engaged up to some moment in time. Another model to describe the process behaviour is based on failures. A failure $f \in \mathit{failures}(P)$ is a pair (s, X) meaning that after the trace $s \in \mathit{traces}(P)$, P refuses all events in X. Finally, failures-divergences extend the failures model with the addition of the process divergences. A divergence of a process is defined as a trace after which the process behaves like Chaos, the most nondeterministic CSP process.

Perhaps the most well-established notion of process inheritance is that defined in [22], in which a process Q is a subprocess of P if the following refinement holds in the failures model: $P \sqsubseteq (Q \setminus (\alpha Q - \alpha P))$, where αP is the alphabet of a process P (set of events in which the process can engage), $S_1 - S_2$ stands for set subtraction, and $P \setminus S$ for a process that behaves as P but hiding the events in the set S. Considering the failures semantics, the previous refinement holds if and only if $\mathit{failures}\ (Q.\mathit{act} \setminus (\alpha Q.\mathit{act} - \alpha P.\mathit{act})) \subseteq \mathit{failures}(P.\mathit{act})$. This notion of inheritance from [22] is the same adopted in *OhCircus*. This is reflected in the obligation that a subprocess main action (its behaviour) must refine, in the failures semantics, the main action (hiding the new events) of its superprocess. In this way the substitutability principle is satisfied. We have actually formally verified this refinement for *Buffer* and *BufferImp* presented in the previous section, as can be found in [8].

A complete account of the *Circus* denotational semantics based on Hoare and He's Unifying Theories of Programming [11] is presented in [15]. As *OhCircus* is a conservative extension of *Circus* we can use the semantics defined in [15] as a basis to formalise the process inheritance notion. So if two processes P and Q have, respectively, $P.act$ and $Q.act$ as their main actions, Q **extends** $P \Leftrightarrow P.act \sqsubseteq_F Q.act \setminus (\alpha Q - \alpha P)$. Here we adopt the same model as that of [22], and consider only failres (not divergences). The reason is that we use hiding in our formulation, and this can introduce divergences, which, in general, makes the failures-divergences refinement fail to hold.

3.1 Semantics of Inheritance

We define a semantics for process inheritance, from which we prove algebraic laws that deal with this feature. Particularly, we define a mapping from processes with inheritance into regular processes, whose semantics is completely defined in [15]. Therefore, it is possible to formally prove the soundness of the proposed set of laws. We give a UTP semantics for a new parallel operator, which turned out to be necessary in the definition of inheritance, as well as for the **super** clause and the **protected** mechanism. Consider the processes *Super* and *Sub* below:

```
process Super
    state st ≙ st₁ ∧ st₂
    pps₁
    pps₂
    • act
end
```

```
process Sub ≙ extends Super
    state st
    pps
    • act
end
```

In the above definition of *Super* we assume that the state st can be split into state schemas st_1 and st_2; these are assumed to be qualified with protected and default visibility mechanisms, respectively. The same visibility considerations are assumed for the schemas $Super.pps_1$ and $Super.pps_2$. In the process given below, $Super.pps_2^{ref}$ is obtained from $Super.pps_2$ by eliminating the paragraphs redefined in $Sub.pps$. Then, given the above considerations, the meaning of Sub is defined as:

$$Sub \mathrel{\widehat=} \left(\begin{array}{l} \mathbf{begin\ state} \mathrel{\widehat=} Super.st \wedge Sub.st \\ \quad Super.pps_{1 \wedge\, \Xi\, Sub.st} \\ \quad Super.pps_2{}^{ref}{}_{\wedge\, \Xi\, Sub.st} \\ \quad Sub.pps \\ \quad \bullet\ Super.act [\![Super.st \mid Super.st \wedge\ Sub.st]\!] Sub.act \\ \mathbf{end} \end{array}\right)$$

In the context of Sub, paragraphs in $Super.pps$ do not modify the state elements in $Sub.st$. The Z schema expression $\Xi Sub.st$ captures this state preservation. The effect of $Super.pps_{1 \wedge\, \Xi\, Sub.st}$ is to ensure that no paragraph in

$Super.pps_1$ modifies state elements in $Sub.st$; the same is true of paragraphs in $Super.pps_2{}^{ref}$. Although all components of $Super$ are in the scope of Sub, only its protected components can be directly accessed by the original declared elements of Sub; as already explained, those with the default qualification cannot be accessed by Sub. Because $Super.act$ can refer to any schema in $Super.pps_1$ or in $Super.pps_2$, and these to any state in $Super.st$, we need to bring all protected and default elements from $Super$ to Sub.

Concerning the main action in the semantics of Sub, it is given by the parallel composition of the main action of Sub with that of $Super$, but we need to impose a protocol concerning access to the shared state elements. This required the definition of a new parallel operator for *OhCircus*, as further explained in the sequel.

3.2 A New Parallel Operator

As originally proposed for *Circus* (and *OhCircus*), the notation for parallel composition of actions A_1 and A_2, synchronising on the channels in the set cs is given by $A_1 [\![ns_1 \mid cs \mid ns_2]\!] A_2$, such that the final state of the variables in ns_1 is given by A_1 and those variables in ns_2 by A_2, with the restriction $ns_1 \cap ns_2 = \emptyset$. It avoids conflicts about what action will determine the final value of a possible shared variable. With this operator, it is not possible to capture the semantics of process inheritance concerning the behaviour of the action in the subprocess. This becomes evident from the main action of Sub, $Super.act [\![Super.st \mid Super.st \wedge Sub.st]\!] Sub.act$ presented above. The restriction that the two sets (ns_1 and ns_2) must be disjoint can be relaxed if we consider that the changes made in a state component by a schema sc in a subprocess cannot contradict the changes made by sc in its superprocess, since the former refines the latter; it follows the same principle described in [12]. Also, note that, in the semantics of process inheritance, we do not need the synchronization set cs, as there is no channel to be shared by a sub and a super process. So our extension is based on a simpler form of parallelism that is actually an interleaving.

Before giving the semantics of the new parallel operator, we introduce some basic notions of the UTP. There are four pairs of observational variables used to define the behaviour of a reactive program in the UTP: the boolean variable $okay$ indicates whether the system has been properly started in a stable state; $okay'$ means subsequent stabilisation in an observable state; tr records the events in which a program has engaged at some moment (tr' records such events at a later moment); the boolean variable $wait$ distinguishes the intermediate observations of waiting states from final observations on termination; in a stable intermediate state, $wait'$ has true as its value (a false value for $wait'$ indicates that the program has reached a final state); all the events that may be refused by a process before the program has started are elements of ref, and possibly refused events at a later moment are referred by ref'. In addition to these observational variables, v and v' stand, respectively, for the initial and intermediate or final values of all program variables.

$$M_{|||} \mathrel{\widehat=} tr' - tr \in (1.tr - tr \mathbin{|||} 2.tr - tr) \land \left(\begin{array}{l} \left(\begin{array}{l} (1.wait \lor 2.wait) \land \\ ref' \subseteq (1.ref \cup 2.ref) \end{array} \right) \\ \lhd wait' \rhd \\ \neg\, 1.wait \land \neg\, 2.wait \land MSt \end{array} \right)$$

$$\begin{array}{l} MSt \mathrel{\widehat=} \forall\, v \bullet (v \in ns_1 \land v \notin ns_2 \\ \quad \Rightarrow v' = 1.v) \\ \land\ (v \in ns_2 \land v \notin ns_1 \Rightarrow v' = 2.v) \\ \land\ (v \in ns1 \cap ns2 \Rightarrow v' = 1.v = 2.v) \\ \land\ (v \notin ns1 \cup ns2 \Rightarrow v' = v) \\ Ui(\{v_1', \ldots, v_n'\}) = i.v_1' = v_1 \land \ldots \\ \quad \land\ i.v_n' = v_n \end{array}$$

Fig. 4. The semantics of our new parallel operator

The formal semantics of the new parallel operator is presented in Figure 4. The merge function $M_{|||}$ is responsible for merging the traces of two actions, and the final values of state components (MSt), local variables and also those of the remaining UTP observational variables; $|||$ takes of two traces and gives a set containing all the possible combinations of them. In $M_{|||}$ the sequence of traces generated by the execution of A_1 and A_2, $(tr' - tr)$ must be a sequence generated by the interleave composition of the traces of A_1 and A_2. The interleaving terminates only if both actions do so. So if $wait'$ is true it is because one of the actions has not finished, $1.wait \lor 2.wait$, and the refusals is contained or equals to the refusals of A_1 and A_2 together. Otherwise if $wait'$ is false it means that both actions has terminated $\neg\, 1.wait \land \neg\, 2.wait$ and the state components and local variables have changed according to the predicate generated by MSt. To avoid name conflicts in the predicate we use a renaming function Ui that prefixes with i the variables in these actions.

This predicate says that each variable in v is changed by A_1 if it belongs uniquely to ns_1, by A_2 if it belongs uniquely to ns_2. If $v \in ns1 \cap ns2$, A_1 and A_2 must agree in the final value of v.

4 Laws

This section presents a small selection of a comprehensive set of algebraic laws for *OhCircus*, particularly addressing specifications with a process hierarchy. The complete set of laws can be found in [8], together with their proofs; these laws range from simple transformations to introduce/eliminate state elements or paragraphs, to more elaborate laws that capture moving elements between super and subprocesses, some of which are presented in this paper. Each law is presented in the form $pds_1 =_{pds} pds_2$, meaning that the set of process declarations pds_1 has the same semantics as the set of process declarations pds_2 in the context of process declarations pds. When a law is valid for any context, we omit the parameter pds. A law may also have a **provided** clause that contains the premises that must be satisfied before its application. As an algebraic law has always two directions of application, we must define the premises for each direction.

Law 1 (process elimination)

$pds\ pd_1 = pds$
where
$pd_1 =$ **process** $P \mathrel{\widehat=}$ [**extends** Q] **begin** $\bullet$ *Skip* **end**
provided
$(\leftrightarrow)\ \neg\ occurs(P, pds)$
$(\leftarrow)\ occurs(Q, pds)$

A process that has its main action as *Skip* (and is not referenced by other processes) does not affect the meaning of a program in *OhCircus*, even if it extends an existing process. We use the notation $occurs(R, pds)$ to represent the fact that the process R is used (as superprocess or in a process composition) by at least one process in pds. For a left to right application of this law, the first proviso guarantees that the process P is not used in pds. For a right to left application, the first proviso ensures that the process declared in pd_1 has a fresh name in pds, whereas the second proviso guarantees that Q must have been previously declared, so that it can be used as a valid superprocess of P. The double arrow in the provided clause means that the condition applies in both directions; otherwise the condition applies only in the direction pointed by the arrow.

Law 2 (super elimination)
This law (see Figure 5) removes the **super** clause from a schema. To remove **super** sc from a schema sc in R, it is necessary that there exists a protected schema sc, in a superprocess of R, as made explicit in the figure. This superprocess must be the closest process to R in its hierarchy. If $P.sc$ has the **super** clause, this is first resolved; as a process hierarchy is a finite structure, it is always possible to find a schema without **super**. The symbol $\oslash$ stands for the Z notation Ξ or Δ. This law is a direct consequence of the semantics of **super** and has no side condition.

Law 3 (splitting a schema among processes)
If part of the behaviour of a schema in a superprocess (including a subset of the state components, related declarations and a predicate) are relevant only for one of its subprocesses, we can introduce a redefinition of this schema in the subprocess and move this part of the original schema to the subprocess as a redefinition of a schema in the superprocess with the remaining part of the original schema.

The state components of P (see Figure 6) are partitioned in two sets st_1 and st_2. $P.sc$, on the right-hand side, changes only st_1, but st_2 is left undefined. $R.sc$ includes $P.sc$ and explicitly constrains the values of the st_2 components according to the predicate $pred_2$; this requires that the state components in this set have the protected access level. Finally there must be no redefinitions of $P.sc$ except in the subprocesses of R.

```
process P ≙ extends Q                      process P ≙ extends Q
    state st                                   state st
    ┌─protected sc                             ┌─protected sc
    │ ⊘st   decls                              │ ⊘st   decls
    ├──────────                                ├──────────
    │ pred                                     │ pred
    └──────────                                └──────────
    pps                                        pps
    • act                                      • act
process R ≙ extends P                =      process R ≙ extends P
    state st                                   state st
    ┌─sc────────                               ┌─sc────────
    │ ⊘st   decls   super sc                   │ ⊘st     ⊘ P.sc.st
    ├──────────                                │ decls   P.sc.decls
    │ pred                                     ├──────────
    └──────────                                │ pred    P.sc.pred
    pps                                        └──────────
    • act                                      pps
                                               • act
```

Fig. 5. super elimination

```
process P ≙ extends Q                         process P ≙ extends Q
    state st₁ ∧ protected st₂                     state st₁ ∧ protected st₂
    ┌─protected sc                                protected sc ≙
    │ ⊘st₁     ⊘ st₂                                 [⊘st₁ decls₁ | pred₁]
    │ decls₁   decls₂                             pps
    ├──────────                                   • act
    │ pred₁    pred₂                          end
    └──────────                        =pds   process R ≙ extends P
    pps                                           state st
    • act                                         protected sc ≙
end                                                  [⊘st₂ decls₂ super sc | pred₂]
process R ≙ extends P                             pps
    state st                                      • act
    pps                                       end
    • act
end
```

provided
($\leftrightarrow$) $\forall S \in pds \mid S \leq P \land \neg\, (S \leq R) \bullet \neg\, occurs(st_2, S.pps) \land \neg\, occurs(st_2, S.act) \land$
$\neg\, impact(st_1, st_2)$
($\rightarrow$) $PL(st_2) \land N(sc) \notin N(R.pps)$

Fig. 6. splitting a schema among processes

In the provided clause in Figure 6, the function N defines the set of process names of a set of process declarations. We overload the function *occurs* in $occurs(sc, R.act)$, $occurs(sc, R.pps)$ and $occurs(sc, R.sc)$. The former represents the fact that the schema *sc* is used in *R.act*; the second, the fact that *sc* is used in *R.pps*; the latter the fact that *sc* is referenced via the **super** clause in *R.sc*. $PL(sc)$ represents the fact that *sc* is a protected schema, and $impact(st_1, st_2)$ is true iff the value of a state component st_1 is affected by the value of st_2.

Law 4 (move action to subprocess)

If the main action of a process P (see Figure 7) can be written as a parallel composition of two actions act_1 and act_2, that access exclusively st_1 and st_2, respectively, we can move one of these actions (in this case, act_2) to a subprocess of P, say R. The state components in st_2 must be protected, so it is possible to refer to them in the R's main action. This law changes the behavior of P, so it cannot be extended by any process in *pds* except for R and its subprocesses

process $P \mathrel{\widehat=}$ **extends** Q
  state $st_1 \wedge st_2$
  pps
  $\bullet\ act_1 [\![st_1 \mid st_2]\!] act_2$
end
process $R \mathrel{\widehat=}$ **extends** P
  state st
  pps
  $\bullet\ act$

$=_{pds}$

process $P \mathrel{\widehat=}$ **extends** Q
  state $st_1 \wedge st_2$
  pps
  $\bullet\ act_1$
end
process $R \mathrel{\widehat=}$ **extends** P
  state st
  pps
  $\bullet\ act [\![st \mid st_2]\!] act_2$

provided
$(\leftrightarrow) \forall S \mid S \in pds \wedge S \neq R \bullet \neg\ occurs(P, S)$
$(\rightarrow) PL(st_2)$

Fig. 7. move action to subprocess

process $P \mathrel{\widehat=}$ **extends** Q
  state $st_1 \wedge st_2$
  pps
  $\bullet\ act$
end
process $R \mathrel{\widehat=}$ **extends** P
  state st
  pps
  $\bullet\ act$

$=_{pds}$

process $P \mathrel{\widehat=}$ **extends** Q
  state st_1
  pps
  $\bullet\ act$
end
process $R \mathrel{\widehat=}$ **extends** P
  state $st \wedge st_2$
  pps
  $\bullet\ act$

provided
$(\leftrightarrow) st_2$ is **protected**
$(\rightarrow)\ \forall S \mid S \leq P \wedge \neg\ (S \leq R) \bullet \neg\ occurs(st_2, S.pps) \wedge \neg\ occurs(st_2, S.act)$
$(\leftarrow)\ \forall S \mid S \leq P \wedge \neg\ (S \leq R) \bullet st_2 \notin PS(S.st)$

Fig. 8. move state component to subproces

(indirectly). Finally, P cannot be used by any process declared in pds, except via inheritance as already mentioned.

Law 5 (move state component to subproces)
A state component st_2 (see Figure 8) of a process P can be moved to one of its subprocesses, say R, if st_2 is not used by P neither by its subprocesses, except those that are also subprocesses of R, including itself. For these, the state component st_2 will be inherited from R instead of P, and no restriction must be applied to them. It must be clear that st_2 is unique through the P process hierarchy. The provisos consider $P.st_2$ as a protected element. The function PS yields, from a set of state components, those in the protected level.

Law 6 (move a protected schema to subprocess)
To move a schema sc (see Figure 9) from P to R, where $R < P$, it is necessary, if sc is protected, that it is not being used by P, neither by its subprocesses, except for those that are also subprocesses of R. Note that we can apply this law, even if a subprocess of P (except for R) has a redefinition of sc.

To move a protected schema $R.sc$ to P, where $R < P$, we must guarantee that neither P nor its subprocesses, except for those that are also subprocesses of R, have a schema named sc.

process $P \mathrel{\widehat{=}}$ **extends** Q		**process** $P \mathrel{\widehat{=}}$ **extends** Q
state st		**state** st
protected sc		pps
pps		$\bullet\ act$
$\bullet\ act$		**end**
end	$=_{pds}$	
process $R \mathrel{\widehat{=}}$ **extends** P		**process** $R \mathrel{\widehat{=}}$ **extends** P
state st		**state** st
pps		**protected** sc
$\bullet\ act$		pps
		$\bullet\ act$

provided
$(\leftrightarrow)\ \forall S \mid S < P \land \neg\,(S < R) \bullet \neg\, occurs(sc, S.pps) \land \neg\, occurs(sc, S.act)$

Fig. 9. move a protected schema to subprocess

Law 7 (subprocess extraction)
In the initial specification of a system it is common to model processes with a very specific behaviour that hides a generic behaviour specialized in face of a particular situation. We propose a law (see Figure 10) that extracts from a process this generic behaviour as a superprocess specializing it with a subprocess. This promotes code reuse and favors a better conceptual representation of the system. The set $R.pps_2'$ in R stands for the schemas in $P.pps_2$ affected by the law, and the set $P.pps_2''$ stands for the updated set $P.pps_2$.

Particularly, it can be proved (see Figure 10) from laws 1, 4, 3, 6, 2 and 5 (LHS stands for the left hand side of the law). First Law 1 is applied creating the process P'; It is easy to observe that P' is equivalent to P. In the next step we apply a double renaming $[P, R/P', P]$ (which clearly preserves the behaviour, since the semantics of P is preserved) followed by Law 4, which moves some elements from R to P. Law 3 is then applied for each schema in $R.pps_2$; the set $R.pps_2'$ stands for the schemas in $R.pps_2$ affected by the law and $P.pps_2''$ for those created in P; as part of this transformation, Laws 6 and 2 are needed when there are protected schemas involved, but we omit these details for conciseness. Finally, Law 5 is applied to move of the unused state components of R to P.

An important issue is a notion of completeness for the proposed set of laws, particulary with respect to inheritance. Our measure for the completeness of the proposed laws is whether their exhaustive application is capable to remove all subprocesses from the target specification. Broadly, by exhaustively applying Law 7, from right to left, we are able to completely remove process inheritance from the specification. In practice, however, it is more common to apply the laws in the opposite direction, since the purpose in design evolution is to introduce (rather than eliminating) inheritance.

5 Case Study

Consider the process *Buffer* as defined in Figure 11. Our intention is to transform this design into a more reusable one, as presented in Section 2.1 (see Figures 2 and 3). The process *Buffer* (in Figure 11) encompasses two abstractions: an

process $P \mathrel{\widehat{=}}$ **extends** Q
  state $st_1 \wedge st_2$
  $pps_{1 \wedge \Xi\ st_2}$
  pps_2
  $\bullet\ act_1 [\![st_1 \mid st_1 \wedge st_2]\!] act_2$
end

$=_{pds}$

process $R \mathrel{\widehat{=}}$ **extends** Q
  state st_1
  pps_1
  pps_2'
  $\bullet\ act_1$
end
process $P \mathrel{\widehat{=}}$ **extends** R
  state st_2
  pps_2''
  $\bullet\ act_2$
end

provided
$(\leftrightarrow)\ R \notin N(pds)$
proof

LHS = [Law 1]

process $P \mathrel{\widehat{=}}$ **extends** Q
  state $st_1 \wedge st_2$
  $pps_{1 \wedge \Xi\ st_2}$
  pps_2
  $\bullet\ act_1 [\![st_1 \mid st_1 \wedge st_2]\!] act_2$
end
process $P' \mathrel{\widehat{=}}$ **extends** P
  $\bullet\ Skip$
end

$= [Law4]$
$[P, R/P', P]$

process $R \mathrel{\widehat{=}}$ **extends** Q
  state $st_1 \wedge st_2$
  $pps_{1 \wedge \Xi\ st_2}$
  pps_2
  $\bullet\ act_1$
end
process $P \mathrel{\widehat{=}}$ **extends** R
  $\bullet\ act_2$
end

=[Law 3]

process $R \mathrel{\widehat{=}}$ **extends** Q
  state $st_1 \wedge st_2$
  $pps_{1 \wedge \Xi\ st_2}$
  pps_2'
  $\bullet\ act_1$
end
process $P \mathrel{\widehat{=}}$ **extends** R
  pps_2''
  $\bullet\ act_2$
end

=[Law 5]

process $R \mathrel{\widehat{=}}$ **extends** Q
  state st_1
  pps_1
  pps_2'
  $\bullet\ act_1$
end
process $P \mathrel{\widehat{=}}$ **extends** R
  state st_2
  pps_2''
  $\bullet\ act_2$
end

Fig. 10. subprocess extraction

abstract unbounded buffer with no concerns about memory space, and a more concrete specialisation that deals with practical memory limitations and offers more functionalities: memory size monitoring and double buffer addition.

Separating these concerns increases reuse and maintainability, and the nature of the design is more faithfully reflected. To achieve these benefits Law 7 (subprocess extraction) can be applied generating the two processes shown in Section 2.1. A key point, before applying the law, is the adaptation of the specification of *BufferImp* to exactly match the left-hand side of Law 7.

Figure 12 shows part of the adaptations we need to perform to apply this law. As a first step, the schemas *Add* and *Remove* are signed as **protected**. Then, the process state is represented as a conjunction of St_1 and St_2; the initialization schema and main action are split accordingly. These transformations are justified by laws of actions, which are not our focus here but can be found in [5]. With

channel $input, input2, output : \mathbb{N}$
channel $length : \mathbb{N}_1$
channel $fc : \mathbb{R}$
channel $start$

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

state St
$queue : \mathrm{seq}\,\mathbb{N}$
$length : \mathbb{N}_1$

$\#queue \leq length$

$Init$
St'
$length? : \mathbb{N}_1$

$queue' = \langle\rangle$
$length' = length?$

Add
ΔSt
$e? : \mathbb{N}$

$queue' = \langle e? \rangle \frown queue$
$\#queue = length \Rightarrow$
$\quad length' = length * 2$

$Remove$
ΔSt
$e! : \mathbb{N}$

$\#queue > 0$
$e! = last\ queue$
$queue' = front\ queue$

$Add2 \mathrel{\widehat=} Add \mathbin{;} Add\,[f?/e?]$

$FactorCapacity$
ΞSt
$fc! : \mathbb{R}$

$fc! = \#queue \ \mathsf{div}\ length$

$Input \mathrel{\widehat=} input?e \to Add$
$Output \mathrel{\widehat=} (\#queue > 0)\&$
$\quad$ **var** $e : \mathbb{N} \bullet Remove;\ output!e \to SKIP$

$Input2 \mathrel{\widehat=} input2?e?f \to Add2$
$Fac \mathrel{\widehat=}$ **var** $fc : \mathbb{N} \bullet FactorCapacity;\ fc!fc \to SKIP$

$\bullet\ start \to length?length \to Init;$
$\mu X \bullet (Input \mathrel{\Box} Output \mathrel{\Box} Input2 \mathrel{\Box} Fac);\ X$
end

Fig. 11. *Buffer* without inheritance

these transformations we can apply Law 7. As explained in the previous section, it embodies several small transformations, resulting in the design in Section 2.1.

6 Conclusions

In this work we proposed a set of sound algebraic laws for *OhCircus*, with focus on process inheritance. As far as we are aware, this is an original contribution, as it seems to be the first systematic characterization of a comprehensive set of laws for process inheritance in the context of rich data types and access control for state and behaviour components. With this goal in mind we started by defining a notion of process inheritance in *OhCircus*. Extending the model of process inheritance [22] for CSP, based on the failures model [10], we defined the semantics for process inheritance in *OhCircus*.

The original design of *OhCircus* makes process components invisible even for its subprocesses, which prevents code reuse. This motivated us to extend the syntax and the semantics of *OhCircus* through the creation of a new access level to signalise the superprocess elements that will be visible to its subprocesses. This also required the definition of typing rules [8] for the new constructs, but we were able to achieve a conservative extension of *OhCircus* [5,15,18], despite the fact that we needed to introduce a new parallel operator, and its UTP semantics, to be able to define the meaning of process inheritance.

We illustrated our overall strategy in a case study where we apply some of the proposed laws. In [8] we address soundness in detail. We have also developed a tool to support our strategy based on the Eclipse Modelling Framework (EMF),

$\textbf{process}\ \mathit{Buffer} \mathrel{\widehat=} \textbf{begin}$

$\textbf{state}\ \mathrel{\widehat=} \mathit{St}_1 \land \mathit{St}_2$

$\ldots$

St_1: $\textbf{protected}\ \mathit{queue} : \operatorname{seq} \mathbb{N}$

St_2: $\mathit{length} : \mathbb{N}_1$ | $\#\mathit{queue} \leq \mathit{length}$

$\textbf{protected}\ \mathit{Add}$: $\Delta \mathit{St}$; $e? : \mathbb{N}$ | $\mathit{queue}' = \langle e? \rangle \frown \mathit{queue}$; $\#\mathit{queue} = \mathit{length} \Rightarrow \mathit{length}' = \mathit{length} * 2$

$\ldots$

$\bullet\ \mathit{start} \to \mathit{Init}_1;$
$\quad \mu X \bullet (\mathit{Input} \mathbin{\Box} \mathit{Output});\ X$
$[\![\mathit{St}_1 \mid \mathit{St}_1 \land\ \mathit{St}_2]\!]$
$\quad \mathit{length}?\mathit{length} \to \mathit{Init}2;$
$\quad \mu X \bullet (\mathit{Input}2 \mathbin{\Box} \mathit{Fac});\ X$
$\textbf{end}$

Fig. 12. *Buffer* adaptations

which was chosen mostly because of the facility for integrating the variety of tools needed, which is archived by the use of a default metamodel, Ecore, across most of EMF technologies. Among EMF tools, we used Xtext for describing the *OhCircus* language, and ATL (Atlas Transformation Language) to encode the algebraic laws and to carry out the mechanised application of the laws. These have not been addressed here for space limitations.

Several works have addressed notions of behavioural subtyping [7,12,16,22] [1,2,3,4]. In [1,12] a subtype relation is defined in terms of invariants over a state, in addition to pre/post conditions and constraint rules over methods. The other cited works define a subtype relation based on models like failures and failures-divergences proposed for CSP, relating refinement with inheritance [22].

In [12] a subtype is allowed add new methods, provided there exists a function that maps these new methods as a combination of the supertype methods; this is not allowed in [22]. Here we allow, in a subtype, new methods like in [12] and even new state components, method overriding, reducing non-determinism, and methods that change both inherited and declared attributes.

Previous works have proposed refinements and algebraic laws for *Circus* [5,18] and these are consequently applicable to *OhCircus*. In [18] the meaning of refinement of processes and their actions are defined based on forward simulation. It also provides an iterative development strategy, involving the application of simulation, action and, most importantly, process refinement. In this context, our work complements [18] with a formal notion of process inheritance and the associated laws.

The mechanization of the formal semantics of *Circus* given in the UTP is provided in [15]. The extension of this work for *OhCircus*, in the form proposed here, is our next immediate goal.

References

1. America, P.: Designing an Object-Oriented Programming Language with Behavioural Subtyping. In: de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1990. LNCS, vol. 489, pp. 60–90. Springer, Heidelberg (1991)

2. Balzarotti, C., Cindio, F., Pomello, L.: Observation equivalences for the semantics of inheritance. In: Proceedings of the IFIP TC6/WG6, FMOODS 1999, Deventer, The Netherlands. Kluwer, B.V. (1999)
3. Bowman, H., Briscoe-Smith, C., Derrick, J., Strulo, B.: On Behavioural Subtyping in LOTOS (1996)
4. Bowman, H., Derrick, J.: A Junction between State Based and Behavioural Specification (Invited Talk), Deventer, The Netherlands, pp. 213–239. Kluwer, B.V. (1999)
5. Cavalcanti, A.L.C., Sampaio, A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Formal Aspects of Computing 15(2-3), 146–181 (2003)
6. Cavalcanti, A.L.C., Sampaio, A., Woodcock, J.C.P.: Unifying Classes and Processes. Software and System Modelling 4(3), 277–296 (2005)
7. Cusack, E.: Refinement, conformance and inheritance. Formal Aspects of Computing 3, 129–141 (1991), doi:10.1007/BF01898400
8. Dihego, J., Antonino, P., Sampaio, A.: Algebraic Laws for Process Subtyping - Extended Version. Technical report (2011), `http://www.cin.ufpe.br/~jdso/technicalReports/TR015.pdf`
9. Fischer, C.: CSP-OZ: A combination of Object-Z and CSP. In: Proceedings of the IFIP, FMOODS 1997, London, UK. Chapman & Hall (1997)
10. Hoare, C.A.R.: Communicating Sequential Processes, vol. 21, pp. 666–677. ACM, New York (1978)
11. Hoare, C.A.R., He, J.: Unifying theories of programming, vol. 14. Prentice Hall (1998)
12. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(6), 1811–1841 (1994)
13. Morgan, C.: Programming from specifications. Prentice-Hall, Inc., Upper Saddle River (1990)
14. Olderog, E.-R., Wehrheim, H.: Specification and (property) inheritance in CSP-OZ. Sci. Comput. Program. 55(1-3), 227–257 (2005)
15. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A UTP Semantics for *Circus*. Formal Aspects of Computing 21(1), 3–32 (2007)
16. Puntigam, F.: Types for Active Objects Based on Trace Semantics. In: Proceedings of the FMOODS 1996, pp. 4–19. Chapman and Hall (1996)
17. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
18. Sampaio, A., Woodcock, J.C.P., Cavalcanti, A.L.C.: Refinement in *Circus*. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 451–470. Springer, Heidelberg (2002)
19. Schneider, S., Treharne, H.: Communicating B Machines. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 416–435. Springer, Heidelberg (2002)
20. Spivey, J.M.: The Z notation: A reference manual. Prentice-Hall, Inc., Upper Saddle River (1989)
21. Toetenel, H., van Katwijk, J.: Stepwise development of model-oriented real-time specifications from action/event models. In: Vytopil, J. (ed.) FTRTFT 1992. LNCS, vol. 571, pp. 547–570. Springer, Heidelberg (1991)
22. Wehrheim, H.: Behavioral Subtyping Relations for Active Objects. Form. Methods Syst. Des. 23(2), 143–170 (2003)