Algebraic Laws for Process Subtyping - Extended Version

José Dihego, Pedro Antonino, and Augusto Sampaio

Universidade Federal de Pernambuco, Centro de Informática,
P.O.Box 7851 50740-540 Recife PE, Brazil
{jdso,prga2,acas}@cin.ufpe.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, with guidelines for FDR verification.

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

1 Introduction

Several formalisms offer support for modelling behavioural and data aspects of a system. For instance, CSP-OZ [11], CSP-B [25], Mosca (VDM+CCS) [27] and *Circus* [18] are some contributions in this direction. Particularly, *Circus* is a combination of Z [26] and CSP [12], which includes constructions in the style of Morgan's refinement calculus [15]. With the intention to also handle object orientation, the *OhCircus* [8] language has been proposed as a conservative extension of *Circus*.

Circus has a refinement calculus that embodies a comprehensive set of laws [7,18,24]. 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 [14]; several works, based on the substitutability principle, have developed theories that recognize suitable inheritance notions between classes [14,28,1]. On the other hand, the semantics of process inheritance is not consolidated. Some of the most well known works about this topic [29,11,16] 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) [13]. 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. In Section 6 we briefly consider the mechanisation of the laws. Finally, in Section 7, we present our conclusions and future work.

Appendix A presents the *OhCircus* process semantics used in proof of our laws. Appendix C list the lemmas used in proofs that change access level. Appendix B explaining how the case study might be verified in the model checker FDR. Appendix D summarizes our laws.

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 processes elements (state and schema operation) by its subprocesses (like the protected mechanism in Java) and the addition of a new clause to define Z schemas [26], very similar to the Z schema inclusion feature, with the aim of allowing schema redefinitions. The **super** clause allows schemas in a subprocess to refer to protected schemas.

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. This

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

Fig. 1. *OhCircus* extended syntax

has motivated the new access level introduced in *OhCircus* to allow code reuse. The keyword **protected** signalises states and schemas belonging to this level.

The syntax for the proposed extensions is presented in Figure 1. 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. Syntactically 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

Buffers are an essential topic in the study of communication protocols, asynchronous systems, the compiling parsing phase, security modeling [20] and buffer tolerance in synchronous systems [21], among other applications. We model the standard concept of an abstract unbounded buffer considering the extensions we

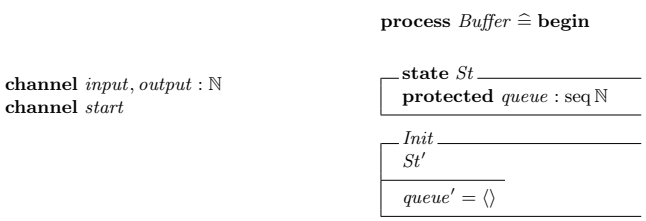


Fig. 2. *Buffer*'s initial and state components

propose to *OhCircus*. Part of the buffer specification is in 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 operations and main action of the buffer is presented in Figure 3. The behaviour of the buffer is to input and output on different channel, 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 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 ($Input \,\Box\, Output$). 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 an implementation of this abstract unbounded buffer, *BufferImp* (see Figure 4). 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

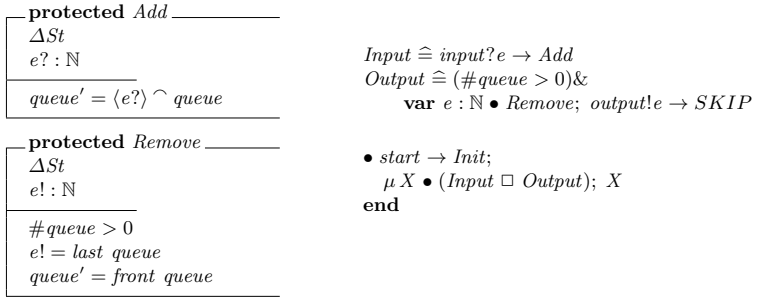


Fig. 3. *Buffer*'s schemas and main action

the ratio between the buffer's size and length[1]. The main action, after initializing the buffer initial length, recursively offers the behaviour $Input2 \ \Box \ Fac$. The local action $Input2$ receives two elements through the channels *input* and *input*2, adding them to the buffer by behaving as $Add2$. The action *Fac* uses the Z schema *FactorCapacity* to inform the ratio size/lenght.

The semantics of process inheritance is given by the parallel composition of the main action of the subprocess with that of its immediate superprocess. The formal details are the subject of the next section. In our example, the semantics of *BufferImp* is given by the interleave composition of its main action with the *Buffer* main action. The schemas *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 processes inheritance.

3 Semantics

Three models to define the behaviour of a CSP process are formally established in [12,22]: traces, failures and failures-divergences. A trace $s \in traces(P)$ 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 $f \in failures(P)$ is a pair (s, X) meaning that after the trace $s \in traces(P)$, 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 [29,16], 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

[1] Note that Ξ is used to indicate that the state is unchanged by the operation, whereas Δ indicates the possibility of state modification

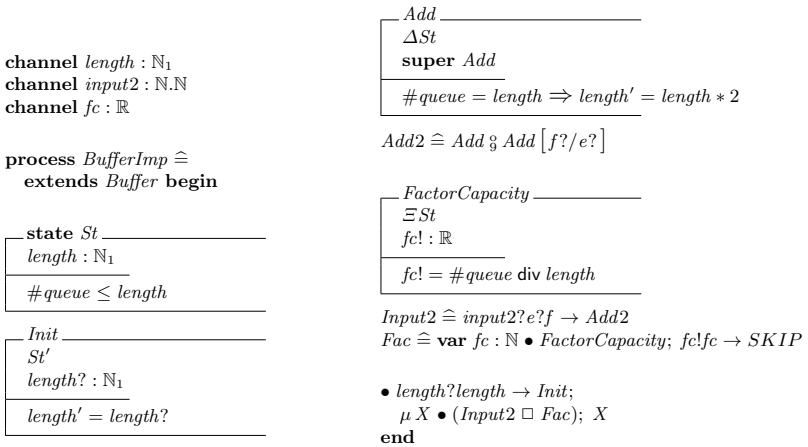


Fig. 4. *BufferImp*: a subprocess of *Buffer*

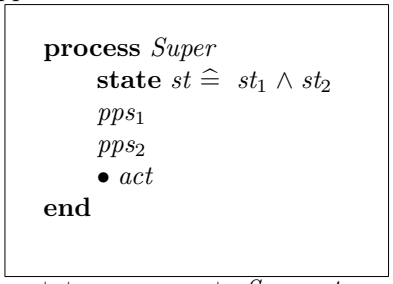
alphabet of the process P (set of events in which the process can engage), $S_1 - S_2$ stands for set subtraction, and $P \setminus S$ a process that behaves as P but hiding the events in the set S. Considering the failures semantics, the previous refinement holds if and only if $failures\ (Q.act \setminus (\alpha Q.act - \alpha P.act)) \subseteq failures(P.act)$. This notion of inheritance from [29] 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 in FDR (see Appendix B).

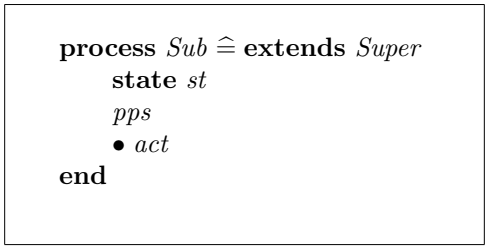
In [18] it is presented a complete account of the *Circus* [18] denotational semantics based on Hoare and He's Unifying Theories of Programming [13]. As *OhCircus* is a conservative extension of *Circus* we can use the semantics defined in [18] as a basis to formalise the process inheritance notion. So if two processes P and Q have, respectively, $P.act$ and $Q.act$ as their main actions, Q **extends** $P \Leftrightarrow P.act \sqsubseteq_F Q.act \setminus (\alpha Q - \alpha P)$. The object-oriented constructs present in *OhCircus* are not addressed in this work since our focus is on process inheritance. Nevertheless, object-oriented constructs are sufficiently independent and do not interfere in the subset of *OhCircus* we are tackling.

3.1 Semantics of Inheritance

We propose the semantics of 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 [18]. 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; see Appendix B for more details.





The state components $Super.st_2$ and $Super.st_1$ 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:

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

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 $Super.pps_{1 \land\ \Xi\ Sub.st}$ 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 UTP Semantics for new Parallel Operator, Visibility and super Clause

An important issue is related to the parallel composition semantics given in [17]. If we have $A_1[\![ns_1 \mid cs \mid ns_2]\!]A_2$, the final state of the variables in ns_1 is

given by A_1 and those variables in ns_2 by A_2, such that $ns_1 \cap ns_2 = \emptyset$. It avoids conflicts about what action will determine the final value of a possible shared variable. Our semantics for process inheritance does not respect this principle. This becomes evident from the main action of *Sub*, $Super.act[\![Super.st|\ Super.st \wedge\ Sub.st]\!]Sub.act$ presented in 3.1. This apparent inconsistency with the semantics of parallel composition can be resolved, 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 [14].

Formally if we consider $ns_1 \cap ns_2 \neq \emptyset$ in $A_1[\![ns_1 \mid ns_2]\!]A_2$ (*cs* empty implies that A_1 and A_2 are composed by interleaving), we are using a new operator, and it arises the obligation of defining its semantics. We define the semantics of this new operator in UTP (Unifying Theories of Programming) [13] based on the semantics given in [17] when $ns_1 \cap ns_2 = \emptyset$. The differences between the current and new interleaving operator are highlighted at the end of this section.

A program is called reactive if its behavior can be observed or even altered at intermediate stable states between initialization and termination [13]. This is the precise case of *OhCircus*. There are four observational variables used to define a program behavior: the Boolean variable *ok* is true for a program that has started in a stable state; its decorated version ok' is true when a program, subsequently, stabilizes in an observable state. The Boolean variable *wait* is true if a program is prepared to engage but is waiting for some event; $wait'$ is true if a program is in a stable intermediate state and false when the program has reached a final state. The variable *tr* records the sequence of events engaged by a process. The variable *ref* records the set of events that a process may refuse before starting; ref' records the set of events that a process may refuse in a stable intermediate state of its execution. The non-observational variables v and v' stand, respectively, for the initial and intermediate values of all program variables.

In the UTP a process is defined as a reactive design of the form $\mathbf{R}(pre \vdash post)$. The design $pre \vdash post$ means $ok \wedge pre \Rightarrow ok' \wedge post$: if a program starts in a state satisfying its preconditions it will terminate and satisfy its postconditions [17]. Using the reactive design we present bellow the formal UTP semantics for $A_1[\![ns_1 \mid ns_2]\!]A_2$, which represents the interleave of the actions A_1 and A_2 considering a possibly non empty intersection between the variables in ns_1 and ns_2.

$$A_1[\![ns_1 \mid ns_2]\!]A_2 \ \widehat{=}\ \mathbf{R}\left(\begin{array}{l} \neg\ {A_1}_f^f \wedge \neg\ {A_2}_f^f \\ \vdash \\ (({A_1}_f^t;\ U1(out\alpha\ A_1)) \wedge ({A_2}_f^t;\ U2(out\alpha\ A_2)))_{\{v,tr\}};\ M_{|||} \end{array}\right)$$

The precondition ${A_1}_f^f \wedge \neg\ {A_2}_f^f$ is a shortcut for $A_1[false/ok'][false/wait] \wedge \neg\ A_2[false/\ ok'][false/wait]$. $A[false/ok'][false/wait]$ represents an action A that diverges when it is not waiting for its predecessor to finish. Since both A_1 and A_2

may execute independently, the interleaving of these actions diverges if either of them diverge [13]. So the precondition guarantees that A_1 and A_2 do not diverge when they are not waiting for their predecessor to finish.

In the postcondition we follow the parallel by merge principle used in [17] and defined in [13]. It executes both actions independently, merging their results when both finish. The notation A_f^t represents an action A that does not diverge when it is not waiting for its predecessor to finish. Consider that ${A_1}_f^t;\ U1(out\alpha\ A_1)$. ${A_1}_f^t$ indicates the execution of A_1 without divergence. $out\alpha\ A$ and $in\alpha\ A$ stand, respectively, for the initial observations of the observational variables in A (undecorated) and for the subsequent observations (eventually the final ones) of A's observational variables (dashed). As we are defining a postcondition, we are interested only in the final values of the observational variables of A_1 and A_2. To avoid name conflicts in the predicate we use a renaming function Ui that prefixes with i the variables in these actions, generating a predicate in the form:

$$Ui(\{v_1', \ldots, v_n'\}) = i.v_1' = v_1 \wedge \ldots \wedge i.v_n' = v_n \tag{1}$$

For example the application of $U1$ to $\{ok', tr'\}$ will generate the predicate $1.ok' = ok \wedge 1.tr' = tr$. Divergence can only happen if it is possible for either of the actions to reach divergence, so the predicate the predicate $(({A_1}_f^t;\ U1(out\alpha\ A_1)) \wedge ({A_2}_f^t;\ U2(out\alpha\ A_2)))$ says that both A_1 and A_2 do not diverge. This predicate is extended with the state components and local variables v and traces tr accompanied by their dashed counterparts. It is expressed using the notation $P_{\{n\}}$, where P is a predicate and n a variable or trace, that means $P \wedge n' = n$.

The last entire predicate is passed to the interleave merge function, which merges the traces of both actions (tr), the state components, local variables (v) and the UTP observational variables ($out\alpha\ A_1$ and $out\alpha\ A_2$), exactly as the parallel merge function found in [17].

$$\begin{aligned} M_{|||} \ &\widehat{=}\ tr' - tr \in (1.tr - tr \ ||| \ 2.tr - tr) \\ &\wedge \begin{pmatrix} \begin{pmatrix} (1.wait \vee 2.wait) \wedge \\ ref' \subseteq (1.ref \cup 2.ref) \end{pmatrix} \\ < wait' > \\ \neg\ 1.wait \wedge \neg\ 2.wait \wedge MSt \end{pmatrix} \end{aligned}$$

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; this operator is defined in [22]. The interleave composition only terminates if both actions do so. So if $wait'$ is true it is because one of the actions has not finished, $1.wait \vee 2.wait$, 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 $\neg\ 1.wait \wedge \neg\ 2.wait$ and the state components and local variables have changed according to the predicate generated by MSt.

$$MSt \mathrel{\widehat=} \forall\, v \bullet (v \in ns_1 \land v \notin ns_2 \Rightarrow v' = 1.v) \land (v \in ns_2 \land v \notin ns_1 \Rightarrow v' = 2.v) \land$$
$$(v \in ns1 \cap ns2 \Rightarrow v' = 1.v = 2.v) \land$$
$$(v \notin ns1 \cup ns2 \Rightarrow v' = v)$$

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. This apparent inconsistency with the semantics of parallel composition is solved, if we consider that the changes made in a state component by schemas in a subprocess cannot contradict the changes made schemas in superprocess.

The new parallel operator is symmetric, associative and distributive. The proof of these properties is similar to the original parallel operator.

The main difference between our interleaving operator and that defined in [17] concerns mainly *MSt*. The original form to this function considers ns_1 and ns_2 disjuncts.

The semantics of our new parallel operator assures that if ns_1 and ns_2 are disjuncts it behaves exactly as the interleave parallel operator defined in [17]. It guarantees that the laws developed for *OhCircus* and *Circus* still valid.

3.3 UTP Semantics for super and protected

The formal meaning of **super** clause, crucial in the proofs of our algebraic laws, must be given, as well as the meaning of the new **protected** access level. We formalize these new constructions in the UTP, as previously done for the new interleave operator.

Schemas in *OhCircus* can be normalized in the same way as schemas in Z. The normalization technique moves all restrictions from the declaration part of a schema to its predicate part. This reduces the declaration part to a canonical form [31]. As an illustration of the schema normalization, consider a process with state *St* and a schema *NextMonth*, as bellow:

St

$month : \mathbb{N}$
$year : \mathbb{N}$

$1 \leq month \leq 12$
$year \geq 2011$

NextMonth

ΔSt

$month' = (month + 1) \bmod 12$
$(month + 1) = 13 \Rightarrow$
$\quad years' = years + 1$

The normalization of *NextMonth* will generate the equivalent schema:

NextMonth

$month, month', year, year' : \mathbb{Z}$

$month \in \mathbb{N} \land month' \in \mathbb{N} \land month' = (month + 1) \bmod 12 \land$
$(1 \leq month \leq 12) \land (1 \leq month' \leq 12) \land$
$year \in \mathbb{N} \land year' \in \mathbb{N} \land ((month + 1) = 13 \Rightarrow years' = years + 1) \land$
$year \geq 2011 \land year' \geq 2011$

The notation ΔSt introduces four state variables ($month, month', year$ and $year'$) of type $\mathbb{N}$, but in the canonical form they assume the more general type $\mathbb{Z}$. Next, this restriction about the type of the four variables is included in the schema predicate, as well as its original schema predicate and the invariant of St.

In [17], the semantics of a Z schema is obtained by transforming it into a statement, whose semantics is given by the following reactive design.

$$w : [pre, post] \mathrel{\widehat{=}} \mathbf{R}\ (pre \vdash\ post \wedge \neg\ wait' \wedge tr' = tr \wedge u' = u)$$

By this reactive design a statement terminates successfully ($\neg\ wait'$), satisfying its postcondition, if its precondition holds. The traces are unchanged ($tr' = tr$) as well as the variables outside w, represented by u ($u' = u$). A transformation of a normalized schema into a statement is given bellow.

$$[udecl;\ ddecl' \mid pred] \mathrel{\widehat{=}}\ ddecl : [\exists\, ddecl' \bullet pred, pred]$$

In a normalized schema the notations for input (?) and output (!) are replaced by undashed ($udecl$) and dashed($ddecl'$) variables, respectively. A predicate ($pred$) determines the effect of the schema. In the statement the variables in $ddecl$ assume a final state that satisfies the predicate ($pred$), the precondition; the predicate itself is the postcondition. As an example consider the meaning of $NextMonth$ in this semantics.

$$month, year :$$
$$\left[\begin{array}{l} \exists\, month', year' : \mathbb{Z} \bullet \\ \left(\begin{array}{l} month \in \mathbb{N} \wedge month' \in \mathbb{N} \wedge month' = (month + 1)\ \mathsf{mod}\ 12 \wedge \\ (1 \leq month \leq 12) \wedge (1 \leq month' \leq 12) \wedge \\ year \in \mathbb{N} \wedge year' \in \mathbb{N} \wedge ((month + 1) = 13 \Rightarrow year' = year + 1) \wedge \\ year \geq 2011 \wedge year' \geq 2011 \end{array}\right), \\ \\ \left(\begin{array}{l} month \in \mathbb{N} \wedge month' \in \mathbb{N} \wedge month' = (month + 1)\ \mathsf{mod}\ 12 \wedge \\ (1 \leq month \leq 12) \wedge (1 \leq month' \leq 12) \wedge \\ year \in \mathbb{N} \wedge year' \in \mathbb{N} \wedge ((month + 1) = 13 \Rightarrow year' = year + 1) \wedge \\ year \geq 2011 \wedge year' \geq 2011 \end{array}\right) \end{array}\right]$$

We extend this definition to deal with schemas having a **super** clause; it formalizes the **super** semantics. Consider the processes $P_1, P_2, \ldots, P_n$, where $P_1 < P_2 < \ldots < P_n$ and the schemas $P_1.sc_1, P_2.sc_2, \ldots, P_n.sc_n$ where sc_1 references sc_2 via **super** clause, which itself references sc_3 and so on; sc_n has no **super** clause. The semantics of sc_1, in the UTP, is given bellow. We consider each sc as a normalized protected schema.

$$sc_1 \mathrel{\widehat{=}} [sc_1.udecl;\ \ldots;\ sc_n.udecl;\ sc_1.ddecl';\ \ldots;\ sc_n.ddecl' \mid sc_1.pred;\ \ldots;\ sc_n.pred]$$

The UTP semantics for **protected** is quite simple since it does not change the original semantics for a schema. Therefore a protected schema sc means exactly what means in default level.

$$\textbf{protected}\ sc \mathrel{\widehat{=}} sc$$

The same is true for a rotected state component. Nevertheless, a protected schema or state component in a superprocess impacts in the meaning of its subprocesses, in any hierarchy level. Consider two processes P and Q, where $Q < P$.

process $P \mathrel{\widehat{=}}$
  state $st \wedge\ [x : T \mid pred]$
  sc
  pps
  $\bullet\ act$
end
process $Q \mathrel{\widehat{=}}$ **extends** P
  state st
  pps
  $\bullet\ act$
end

The meaning of Q, in the UTP, is given by:

$$Q \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{begin state} \mathrel{\widehat{=}} P.st \wedge [x : T \mid pred] \wedge Q.st \\ \quad P.pps_1 \wedge sc_{\wedge\ \Xi\ Q.st} \\ \quad {P.pps_2}^{ref}{}_{\wedge\ \Xi\ Q.st} \\ \quad Q.pps \\ \quad \bullet\ P.act[\![P.st \wedge P.x \mid P.st \wedge P.x \wedge\ Q.st]\!] Q.act \\ \textbf{end} \end{array} \right)$$

where ${P.pps_2}^{ref}$ are obtained removing the elements redefined in $Q.pps$. The schemas in $Q.pps_1$ have not the **super** clause and cannot access the default state components $P.st_1 \wedge P.x$. This is possible for schemas in $Q.pps_2$, which have the **super** clause. If we change the above specification to:

$$
\begin{array}{l}
\textbf{process } P \mathrel{\widehat{=}} \\
\quad \textbf{state } st \wedge \ [\textbf{protected } x : T \mid pred] \\
\quad \textbf{protected } sc \\
\quad pps \\
\quad \bullet \ act \\
\textbf{end} \\
\textbf{process } Q \mathrel{\widehat{=}} \textbf{extends } P \\
\quad \textbf{state } st \\
\quad pps \\
\quad \bullet \ act \\
\textbf{end}
\end{array}
$$

The meaning of Q in UTP becomes:

$$
Q \ \widehat{=} \left(\begin{array}{l}
\textbf{begin state } \widehat{=} \ P.st \wedge [\textbf{protected } x : T \mid pred] \wedge Q.st \\
\quad P.pps_1 {}_{\wedge \ \Xi \ Q.st} \\
\quad P.pps_2 {}^{ref}_{\wedge \ \Xi \ Q.st} \\
\quad \textbf{protected } sc \\
\quad Q.pps \\
\quad \bullet \ P.act[\![P.st \wedge P.x \mid P.st \wedge P.x \wedge \ Q.st]\!] Q.act \\
\textbf{end}
\end{array} \right)
$$

As $P.sc$ is now a protected schema, it can be overridden by Q, therefore the addition of $\Xi \ Q.st$ to this schema is removed. Changing $P.x$ to protected allows all schemas, including the main action of Q to access it directly.

4 Algebraic Laws

This section presents a set of soundness algebraic laws for *OhCircus*. These laws address specifications with a process hierarchy. As far as we are aware, this is an original contribution of this work, as it seems to be the first systematic characterization of a comprehensive set of laws for process inheritance that use rich data types and access control for state components and behavior components (Z schemas).

The proposed laws act on the process hierarchy of a specification, as well as on the state components and schemas of a process. Laws of actions for *Circus*, proposed in [6], are also valid for *OhCircus*, so this is not the focus in this work.

The laws are classified into two groups. Simple laws are justified directly from the semantics, whereas composite laws are proved to be a consequence of other laws, simple or composed laws.

Each law may have a **provided** clause that contains the premisses that must be satisfied before its application. As an algebraic law has always two directions of application, we must define the premisses for each direction.

Consider two processes P and Q. If P is refined by Q ($P \sqsubseteq Q$) the main action of P is refined by that of Q ($P.act \sqsubseteq Q.act$). From the semantics of process inheritance, if $Q < P$, the state components of Q are those declared in Q and those inherited from P. The same rule is applied to Z schemas. As process inheritance is a transitive relation, the elements inherited from P are those declared in P and those inherited from its immediate superprocess, and so on. Differently from state components and schemas, actions cannot be inherited, so any action defined in P is hidden from Q. Although the actions are not eligible for inheritance, the behavior of the main action of a subprocess Q is given by that declared in Q in interleave with that **representing** (not necessarily the declared action) the behavior of its immediate superprocess. The reason is that the superprocess P might itself inherit from another process.

The behavior of a process is not given only by its declared main action. For a subprocess Q there is an implicity main action, $Q.\overline{act}$, formed by the interleave of its declared main action $Q.act$ and the implicit main action $P.\overline{act}$ of its immediate superprocess P. If P is at the top of the inheritance hierarchy, then $P.\overline{act} = P.act$, otherwise it follows the same reasoning applied for Q.

If we have $Q < P$, it implies that $P.\overline{act} \sqsubseteq Q.\overline{act}$. This implication is used in the proof of correctness of our laws. As already mentioned, the laws of actions are the subject of [6]. We use such laws to justify some of the laws we propose for processes involved in an arbitrary inheritance hierarchies.

4.1 The semantics of *Circus* processes

In accordance with [17] (see Appendix A) a process meaning is given by a command. Consider the process P as bellow:

$$P \mathrel{\widehat=} \left(\begin{array}{l} \textbf{begin state} \mathrel{\widehat=} P.st \\ \quad P.pps \\ \quad \bullet\ P.act \\ \textbf{end} \end{array}\right)$$

The meaning of such a process is the command:

$$\textbf{var} P.st.decl \bullet ER(P.act, P.st.inv, P.pps) \tag{2}$$

which, also by [17], is defined by the existential quantification:

$$\exists\, P.st.decl, P.st.decl' \bullet ER(P.act, P.st.inv, P.pps) \tag{3}$$

In the above predicate $P.st.decl$ denotes the declaration part of $P.st$. Its invariants $P.st.inv$ are conjoined with the schemas $P.pps$ and in the main action $P.act$, in the form $\textbf{if} P.st.inv \rightarrow Skip [\!] Stop$, after each atomic component action.

Finally all occurrences of schemas $P.pps$ (conjoined with $P.st.inv$) in the main action are replaced by semantically equivalent commands (specification statements) following the semantics for schemas presented in [17]. All this work is done by the enforce-replace (ER) function.

$$ER : \mathsf{Action} \pfun \mathbb{P}\,\mathsf{Constraint} \pfun \mathbb{P}\,\mathsf{PParagraph} \pfun \mathsf{Action}$$
$$ER\ a\ \emptyset\ \emptyset = a$$
$$ER\ a\ \emptyset\ p = R(a, p)$$
$$ER\ a\ inv\ \emptyset = E(a, inv)$$
$$ER\ a\ i : invs\ p : pps = ER(E(R(a, E(p, i)), i), invs, pps)$$

When this function receives an action and empty sets of constraints and paragraphs returns the action unchanged. If the set of invariants is empty, the function R replace the occurrences of a given schema p in the action a by its equivalent specification statement (see Subsection 3.3). When the set of schemas is empty, the function E enforces the action a in order to guarantee the invariant inv. In the general case, the function recursively enforces all paragraphs with the invariants, replaces them in the action, and enforces this resulting action with the invariants.

$$E : \mathsf{Action} \pfun \mathsf{Constraint} \pfun \mathsf{Action}$$
$$E\ a\ inv = a;\ \mathbf{if} inv \rightarrow Skip [\!] Stop$$
$$E\ (a\ op\ b)\ inv = E(a, inv)\ op\ E(b, inv)$$

The function E receives an action and an invariant, if the action a is atomic E returns a; $\mathbf{if} inv \rightarrow Skip [\!] Stop$. Otherwise if the action can be written as a composition of other actions ($a\ op\ b$), using a proper operator op, the function enforces each action and composes the result actions with op. We overload this function to schemas. It receives a schema and an invariant and returns a schema whose predicate is conjoined with the invariant.

$$E : \mathsf{Paragraph} \pfun \mathsf{Constraint} \pfun \mathsf{Paragraph}$$
$$E\ [udecl;\ ddecl' \mid pred]\ inv = [udecl;\ ddecl' \mid pred \land inv]$$

Finally the function R replaces all occurrences of a schema in an action by its equivalent specification statement. If an atomic action a is just a call for a schema sc ($a = sc$ with a small abuse of notation), it can be replaced by sc equivalent specification statement, otherwise a is returned unchanged. If the action can be written as a composition of other actions ($a\ op\ b$), using a proper operator op, the function replaces in each action all occurrences of sc by its equivalent specification statement and composes the resulting actions with op.

$$R : \mathsf{Action} \pfun \mathsf{Paragraph} \pfun \mathsf{Action}$$
$$R\ a\ sc \mathrel{\widehat=} [udecl;\ ddecl' \mid pred] = \mathbf{if}\ a = sc\ \mathbf{then}\ [udecl;\ ddecl' \mid pred]\ \mathbf{else}\ a$$
$$R\ (a\ op\ b)\ sc = R(a, sc)\ op\ R(b, sc)$$

To prove some of the laws presented in this section we need to work with the semantics of a process at the predicate level. These laws, excepting the Law

3, are proved from the UTP semantics for process inheritance, based on [17]. Since [17] does not present the semantics for a program, we use in the Law 3 the semantics developed in [30] for a *Circus* program. Considering the absent of object-orientation constructs we can apply this semantics to a program in *OhCircus*, where process inheritance was removed by our semantics developed in Section 3. It is not our intent to provide a possible theory of equivalence between [30] and [17].

4.2 Laws

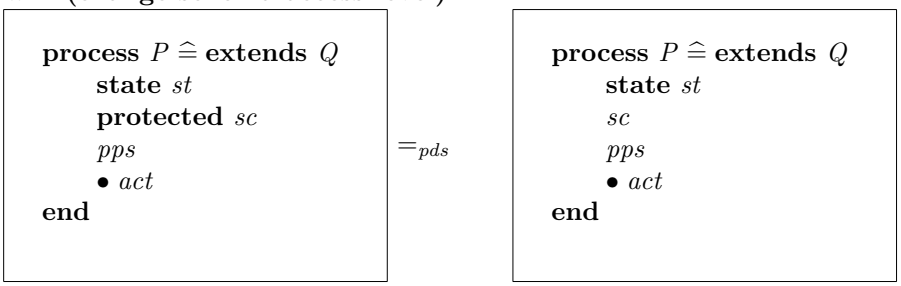
We categorize the laws in three groups: localized eliminations, access modifications and element interchanges between processes related by inheritance. The laws in the first group insert or remove elements of a process considering its hierarchy. In the second group we have the laws that change the access modifiers of state components and schemas. The latter groups laws responsible for interchange elements between a process and one of its subprocess.

Access Modifications

To change the access level of a schema sc in P from protected to default we must guarantee that all superprocesses of P do not declare a schema homonymous to sc, or if it happens, this schema must have the default access level. Note that it is not possible to have a default schema in a subprocess with the same name as one in the protected level in any of its superprocesses; therefore we do not need explicit side conditions to capture this. In addition, all subprocesses of P cannot redefine sc.

The provisos in the reverse direction of the law application guarantee that the superprocesses of P have not a schema homonymous to sc. According to our typing rules a default schema cannot redefine a protected one. Furthermore, the subprocesses of P have not a schema homonymous to sc. The typing rule mentioned motivates this restriction.

Law 1 (change schema access level).



provided

$(\rightarrow)$ $(1) \forall Q \mid P < Q \bullet (\forall s \in Q.pps \mid N(s) = N(sc) \rightarrow \neg PL(s))$
$(2) \forall R \mid R < P \bullet \neg occurs(sc, R.act) \wedge \forall s \in R.pps \bullet (\neg N(sc) = N(s) \wedge \neg occurs(sc, s))$
$(\leftarrow)$ $(1) \forall Q \mid P < Q \bullet sc \notin P.pps$ $(2) \forall R \mid R < P \bullet sc \notin R.pps$

proof

The meaning of P on the left-hand side is defined as:

$$P \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ P.st} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ P.st} \\ \quad P.pps \\ \quad \textbf{protected}\ P.sc \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ \ P.st]\!]P.act \\ \textbf{end} \end{array}\right)$$

= [By Lemma 1]

$$P \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ P.st} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ P.st} \\ \quad P.pps \\ \quad P.sc \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ \ P.st]\!]P.act \\ \textbf{end} \end{array}\right)$$

That is the exact meaning of P, in the UTP, on the right-hand template.

A subprocess of P, lets say R, in *pds* means:

$$R \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st \wedge R.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ P.st \wedge R.st} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ P.st \wedge R.st} \\ \quad P.pps_1{}_{\wedge\ \Xi\ R.st} \\ \quad P.pps_2{}^{ref}_{\Xi\ R.st} \\ \quad \textbf{protected}\ P.sc \\ \quad R.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ \ P.st]\!]P.act \\ \qquad [\![Q.st \wedge P.st \mid Q.st \wedge P.st \wedge R.st]\!] \\ \quad R.act \\ \textbf{end} \end{array}\right)$$

= [By Lemma 1]

$$R \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{begin state} \mathrel{\widehat{=}} Q.st \wedge P.st \wedge R.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ P.st \wedge R.st} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ P.st \wedge R.st} \\ \quad P.pps_1{}_{\wedge\ \Xi\ R.st} \\ \quad P.pps_2{}^{ref}_{\Xi\ R.st} \\ \quad P.sc \\ \quad R.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!]P.act \\ \qquad [\![Q.st \wedge P.st \mid Q.st \wedge P.st \wedge R.st]\!] \\ \quad R.act \\ \textbf{end} \end{array}\right)$$

The above process is the exact meaning of R, in the UTP, on the right-hand template. This can be generalized for each subprocess of P, in any hierarchy level. This concludes our proof. The application of this law in the reverse direction is similar and we omit it here.

□

To reduce the access level of a state component st_1 in P, from protected to default level, it cannot be used by schemas nor actions of P's subprocesses. As the superprocesses of P do not even know that P exists and the overriding of state components is not allowed, no restrictions are applied to them.

A protected state component is unique in a process hierarchy, so if a process declares a protected state component st_1 neither of its super/subprocesses can have one homonymous to st_1. The proviso for the right application of the law guarantees that the state component st_1 is unique in the P's hierarchy. The uniqueness for protected state components does not apply for those with default access level. Therefore it is perfectly possible to have a default state component $A.st_1$ and a protected $B.st_1$, where $B < A$, as $A.st_1$ is not visible in B. Otherwise if $A.st_1$ is protected, it is visible in B and we have that $st_1 \notin B.st$.

Law 2 (change state component access level).

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

provided
$(\rightarrow)\ \forall R \mid R < P \bullet \neg\ occurs(R.st_1, R.act) \wedge \neg\ occurs(R.st_1, R.pps)$
$(\leftarrow)\ \forall R \mid R < P \bullet st_1 \notin PS(R.st)\ \wedge\ \forall Q \mid P < Q \bullet st_1 \notin PS(Q.st)$

proof

The meaning of P in the left-hand side is defined as:

$$P \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state } \widehat{=}\; Q.st \wedge P.st \wedge \textbf{protected } st_1 \\ \quad Q.pps_{1\wedge\ \Xi\ P.st\wedge P.st_1} \\ \quad Q.pps_2{}^{ref}{}_{\wedge\ \Xi\ P.st\wedge P.st_1} \\ \quad P.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ \ P.st \wedge P.st_1]\!]P.act \\ \textbf{end} \end{array}\right)$$

= [By Lemma 2]

$$P \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state } \widehat{=}\; Q.st \wedge P.st \wedge\ \ st_1 \\ \quad Q.pps_{1\wedge\ \Xi\ P.st\wedge P.st_1} \\ \quad Q.pps_2{}^{ref}{}_{\wedge\ \Xi\ P.st\wedge P.st_1} \\ \quad P.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ \ P.st \wedge P.st_1]\!]P.act \\ \textbf{end} \end{array}\right)$$

The above process is the exact meaning of P, in the UTP, on the right-hand template.

A subprocess of P, lets say R, in *pds* means:

$$R \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state } \widehat{=}\; Q.st \wedge P.st \wedge \textbf{protected } P.st_1 \wedge R.st \\ \quad Q.pps_{1\wedge\ \Xi\ P.st\wedge P.st_1\wedge R.st} \\ \quad Q.pps_2{}^{ref}{}_{\wedge\ \Xi\ P.st\wedge P.st_1\wedge R.st} \\ \quad P.pps_{1\wedge\ \Xi\ R.st} \\ \quad P.pps_2{}^{ref}{}_{\Xi\ R.st} \\ \quad R.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ \ P.st \wedge P.st_1]\!]P.act \\ \quad\quad [\![Q.st \wedge P.st \wedge P.st_1 \mid Q.st \wedge P.st \wedge P.st_1 \wedge R.st]\!] \\ \quad R.act \\ \textbf{end} \end{array}\right)$$

= [By Lemma 2]

$$
R \mathrel{\widehat{=}} \left(\begin{array}{l}
\textbf{begin state} \mathrel{\widehat{=}} Q.st \wedge P.st \wedge P.st_1 \wedge R.st \\
\quad Q.pps_{1\wedge\ \Xi\ P.st \wedge P.st_1 \wedge R.st} \\
\quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ P.st \wedge P.st_1 \wedge R.st} \\
\quad P.pps_{1\wedge\ \Xi\ R.st} \\
\quad P.pps_2{}^{ref}_{\Xi\ R.st} \\
\quad R.pps \\
\quad \bullet\ Q.act [\![Q.st \mid Q.st \wedge \ P.st \wedge P.st_1]\!] P.act \\
\qquad [\![Q.st \wedge P.st \wedge P.st_1 \mid Q.st \wedge P.st \wedge P.st_1 \wedge R.st]\!] \\
\quad R.act \\
\textbf{end}
\end{array}\right)
$$

The above process is the exact meaning of R, in the UTP, on the right-hand template. This can be generalized for each subprocess of P, in any hierarchy level. This concludes our proof. The application of the this law in the reverse direction is similar.

□

Localized Eliminations

The meaning of a program in *OhCircus* is given by the meaning of each process and class declaration. Furthermore, global Z schema, constant, channel, and channel set definitions [30] are also taken into account.

A process having its main action as *Skip* does not affect the meaning of a program in *OhCircus*. In addition it does not have a superprocess, but can be one, since the meaning of its subprocesses remains unchanged with or without the inheritance relation.

We use the notation $occurs(P, pds)$ to represent the fact that the process P is used (as superprocess or in a process composition) by at least one process in pds. The function N defines the set of process names of a set of process declarations.

On the right-hand side of this law, the proviso guarantees that the process P is not used in pds. On the left-hand side the process declared in pd_1 has a fresh name in pds.

Law 3 (process elimination).
$pds\ pd_1 = pds$

where
$pd_1 =$ **process** $P \mathrel{\widehat{=}}$ **begin** $\bullet$ *Skip* **end**

provided
$(\leftrightarrow)\ \neg\ occurs(P, pds)$

proof
This law can be directly justified from the semantics of programs given in [30], where a program is defined as a conjunction of the semantics of each component process. Therefore in the semantics of $pds\ pd_1$ bellow, $[\![prog]\!]^{PROG}$ stands for the semantics of a program *prog* and $[\![p]\!]^{P}$ for the semantics of a process p. The semantics is defined as a relation where *Skip* is mapped into an empty relation.

$[\![pds\ pd_1]\!]^{PROG}$
= [From the semantics of programs]
$[\![pds]\!]^{PROG}[\![pd_1]\!]^{P}$
= [From the assumption]
$[\![pds]\!]^{PROG}\ \emptyset$
= [*Skip* is the empty relation]
$[\![pds]\!]^{PROG}$

□

A default schema $P.sc$ can be eliminated from P when it is no longer referenced by P' schemas or used by its actions. No restrictions are applied to its superprocesses or subprocesses. This is justified by the fact that a default schema cannot be inherited and, consequently, it cannot be redefined. $PL(sc)$ represents the fact that sc is a protected schema.

The addition of a default schema sc in P is allowed if sc is distinct from those schemas declared in P and if in all superprocesses of P there is not a protected schema with the same name as sc. The latter condition must hold because a default schema cannot refine a protected one. There are no restrictions about the subprocesses of P since the default elements of P are hidden.

Law 4 (default schema elimination).

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

$=_{pds}$

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

provided
$(\rightarrow)\ \neg\ occurs(P.sc, P.act) \wedge \neg\ occurs(P.sc, P.pps)$
$(\leftarrow)\ sc \notin P.pps \wedge (\forall\ Q \mid P < Q \bullet sc \notin Q.pps, \mathbf{if} PL(sc))$
proof
The semantics of P on the left-hand side is defined as:

$$P\ \widehat{=}\ \left(\begin{array}{l}\mathbf{begin\ state}\ \widehat{=}\ Q.st \wedge P.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ P.st} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ P.st} \\ \quad P.pps \\ \quad P.sc \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!]P.act \\ \mathbf{end}\end{array}\right)$$

= [By A.1]

$$\mathbf{var} Q.st.decl \wedge P.st.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!] \quad (4)$$
$$P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps \wedge P.sc)$$

= [By A.3]

$$\exists\ Q.st.decl \wedge P.st.decl, Q.st.decl' \wedge P.st.decl' \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st \quad (5)$$
$$P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps \wedge P.sc)$$

In the above predicate $Q.st.decl$ denotes the declaration part of $Q.st$; the same applies to $P.st.decl$ (their dashed values represents variables final values). Their invariants, respectively $Q.st.inv$ and $P.st.inv$ are conjoined with the schemas $Q.pps \wedge P.pps \wedge P.sc$ and enforced in the main action $Q.act[\![Q.st \mid Q.st \wedge P.st]\!]P.act$ by the ER function.

Finally all occurrences of schemas $Q.pps \wedge P.pps \wedge P.sc$ (conjoined with $Q.st.inv \wedge P.st.inv$) in the main action are replaced by semantically equivalent commands (specification statements) following the semantics for schemas presented in [17]. All this work is done by the enforce-replace (ER) function.

Considering an action A, $ER(A, inv, pps \wedge sc) = ER(A, inv, pps)$ if $\neg\ occurs(sc, A)$ and $\neg\ occurs(sc, pps)$. Considering this and the provisos we have that the semantics of P is:

$$\exists\ Q.st.decl \wedge P.st.decl, Q.st.decl' \wedge P.st.decl' \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st \quad (6)$$
$$P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps)$$

= [By A.3]

$$\mathbf{var} Q.st.decl \wedge P.st.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge \ P.st]\!] \quad (7)$$
$$P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps)$$

= [By A.1]

$$P \ \widehat{=} \begin{pmatrix} \mathbf{begin\ state} \ \widehat{=} \ Q.st \wedge P.st \\ \quad Q.pps_1 {}_{\wedge \ \Xi \ P.st} \\ \quad Q.pps_2 \overset{ref}{}_{\wedge \ \Xi \ P.st} \\ \quad P.pps \\ \quad \bullet \ Q.act[\![Q.st \mid Q.st \wedge \ P.st]\!] P.act \\ \mathbf{end} \end{pmatrix}$$

The above process is the exact meaning of P, in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar and we omit it here.

□

To remove a default state component st_1 from P it is necessary that st_1 be not used by actions nor schemas of P. No restrictions are applied to the subprocesses of P since st_1 is a default schema and cannot be inherited.

The insertion of default state component st_1 in P is conditional to the absence of a state component in P homonymous to st_1. No restrictions are applied to super/subprocesses of P since default state components can appear many times in a process hierarchy, whereas protected state components are unique.

Law 5 (default state component elimination).

process $P \ \widehat{=}$ **extends** Q **state** $st \wedge st_1$ pps $\bullet$ act **end**	$=_{pds}$	**process** $P \ \widehat{=}$ **extends** Q **state** st pps $\bullet$ act **end**

provided
$(\rightarrow)$ $\neg$ $occurs(P.st_1, P.act) \wedge \neg$ $occurs(P.st_1, P.pps)$
$(\leftarrow)$ $st_1 \notin P.st$

proof
The semantics of P in the left-hand side is defined as:

$$P \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st \wedge P.st_1 \\ \quad Q.pps_1 {}_{\wedge\ \Xi\ P.st \wedge P.st_1} \\ \quad Q.pps_2 {}^{ref}_{\wedge\ \Xi\ P.st \wedge P.st_1} \\ \quad P.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge P.st \wedge P.st_1]\!]P.act \\ \textbf{end} \end{array}\right)$$

= [By A.1]

$$\textbf{var}\, Q.st.decl \wedge P.st.decl \wedge P.st_1.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge P.st \wedge P.st_1]\!] \quad (8)$$
$$P.act, P.st.inv \wedge P.st_1.inv \wedge Q.st.inv, Q.pps \wedge P.pps)$$

= [By A.3]

$$\exists\, Q.st.decl \wedge P.st.decl \wedge P.st_1.decl \wedge Q.st.decl' \wedge P.st.decl' \wedge P.st_1.decl' \quad (9)$$
$$\bullet\ ER(Q.act[\![Q.st \mid Q.st \wedge P.st \wedge P.st_1]\!]P.act,$$
$$P.st.inv \wedge P.st_1.inv \wedge Q.st.inv, Q.pps \wedge P.pps)$$

Considering that $\exists\ x;\ x', y, y' : T \bullet A = \exists\ x;\ x' : T \bullet A$ if y, y' do not occur in A; and for an atomic action act, $act \rightarrow \textbf{if}\, inv \rightarrow Skip[\!]Stop = act$ if inv is a predicate about a variable not used in act (this implies that before and after act execution, inv has the same evaluation), the existential quantification, considering also the provisos, becomes:

$$\exists\, Q.st.decl \wedge P.st.decl \wedge Q.st.decl' \wedge P.st.decl' \quad (10)$$
$$\bullet\ ER(Q.act[\![Q.st \mid Q.st \wedge P.st]\!]P.act,$$
$$P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps)$$

= [By A.3]

$$\textbf{var}\, Q.st.decl \wedge P.st.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge P.st]\!] \quad (11)$$
$$P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps)$$

= [By A.1]

$$P \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st \\ \quad Q.pps_1{}^{\wedge\ \Xi\ P.st} \\ \quad Q.pps_2{}^{ref\ \wedge\ \Xi\ P.st} \\ \quad P.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge P.st]\!]P.act \\ \textbf{end} \end{array}\right)$$

The above process is the exact meaning of P, in the UTP, on the right-hand template. This concludes our proof. The proof of the this law in the reverse direction is similar and we omit it here.

□

The insertion of a protected state component st_1 in P is possible if its super-processes and subprocesses, including P itself, do not declare a protected state component homonymous to st_1. The function PS determines, from a set of state components, those in the protected level.

The provisos for the right-hand side application of the law guarantee that the protected state component st_1 is not used by P nor by its subprocesses.

Law 6 (protected state component elimination).

process $P \,\widehat{=}$ **extends** Q **state** $st \wedge$ **protected** st_1 pps $\bullet\ act$ **end**	$=_{pds}$	**process** $P \,\widehat{=}$ **extends** Q **state** st pps $\bullet\ act$ **end**

provided
$(\rightarrow)\ \forall R \mid R \leq P \bullet \neg\ occurs(R.st_1, R.act) \wedge \neg\ occurs(R.st_1, R.pps)$
$(\leftarrow)\ \forall R \mid R \leq P \bullet st_1 \notin PS(R.st)\ \wedge\ \forall Q \mid P < Q \bullet st_1 \notin PS(Q.st)$

proof
The proof of this law is a consequence of Law 2 and Law 5. We present the left-hand side proof; the right-hand side is similar.

process $P \mathrel{\widehat{=}}$ **extends** Q
  state $st \wedge$
    protected st_1
  pps
  • act
end

= [By Law 2 application]

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

= [By Law 5 application]

process $P \mathrel{\widehat{=}}$ **extends** Q
  state st
  pps
  • act
end

□

The insertion of a protected schema sc in P is possible if its superprocesses and subprocesses, including P itself, do not have a schema with the same name as sc. We abuse the notation when we write $sc \in pps$; in fact, it is true if the name of schema sc is used by at least one schema in pps.

In the case where a subprocess of P has a schema with the same name as that of sc, this must refine sc. Likewise, if a superprocess of P has an homonymous schema to sc, this schema must be refined by sc. Therefore, there is a restriction about schema overriding: if a schema is overridden in a subprocess the specialized version must maintain the same level of access as the original version, obviously, the protected level.

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 $P.pps$; the latter the fact that sc is referenced via the **super** clause in $R.sc$.

The proviso, on the right-hand side of the law, guarantees that the schema sc is not used by any subprocess of P, neither by P itself. If it is the case, sc can be removed from P. This condition is needed since sc is in the protected level and all subprocesses of P inherits it. If a subprocess of P redefines sc without including it using the **super** clause, the removal of sc from P is still valid.

Law 7 (protected schema elimination).

process $P \mathrel{\widehat{=}}$ **extends** Q
  state st
  protected sc
  pps
  • act
end

$=_{pds}$

process $P \mathrel{\widehat{=}}$ **extends** Q
  state st
  pps
  • act
end

provided
$(\rightarrow)$ $\forall R \mid R < P \bullet \neg\, occurs(sc, R.act) \wedge \neg\, occurs(sc, R.pps) \vee$
$N(sc) \in N(R.pps) \Rightarrow P.sc \sqsubseteq R.sc \wedge \neg\, occurs(sc, R.sc)$
$(\leftarrow)$ $sc \notin P.pps \wedge (\forall R \mid R < P \bullet sc \notin R.pps \vee (sc \in R.pps \wedge P.sc \sqsubseteq R.sc)) \wedge$
$(\forall Q \mid P < Q \bullet sc \notin Q.pps \vee (sc \in Q.pps \wedge Q.sc \sqsubseteq P.sc))$

proof
The proof of this law is a consequence of Law 1 and Law 4. We present the left-hand side proof; the right-hand side is similar.

process $P \mathrel{\widehat{=}}$ **extends** Q
  state st
  protected sc
  pps
  • act
end

= [By Law 1 application and its provisos]

process $P \mathrel{\widehat{=}}$ **extends** Q
  state st
  sc
  pps
  • act
end

= [By Law 4 application and its provisos]

process $P \mathrel{\widehat=}$ **extends** Q
  state st
  pps
  • act
end

□

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 in the bellow definition. 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 ⊘ stands for the Z notation Ξ or Δ.

Law 8 (super elimination).

process $P \mathrel{\widehat=}$ **extends** Q
  state st

  protected sc
  ⊘st
  $decls$

  $pred$

  pps
  • act

process $R \mathrel{\widehat=}$ **extends** P
  state st

  sc
  ⊘st
  $decls$
  super sc

  $pred$

  pps
  • act

$=_{pds}$

process $P \mathrel{\widehat=}$ **extends** Q
  state st

  protected sc
  ⊘st
  $decls$

  $pred$

  pps
  • act

process $R \mathrel{\widehat=}$ **extends** P
  state st

  sc
  ⊘st
  ⊘$P.sc.st$
  $decls$
  $P.sc.decls$

  $pred$
  $P.sc.pred$

  pps
  • act

proof
This law is a direct consequence of the semantics of **super**, which is given in the process level.

□

Whenever there is a protected schema sc in a superprocess of P, it is possible to define in P a protected schema $P.sc$, whose body is composed uniquely by a **super** clause referring to sc (see **super** semantics in subsection 3.3). $P.sc$ is a trivial redefinition of sc. This trivial redefinition can be removed, no matter the context.

Law 9 (eliminating a trivial schema redefinition).

process $Q \mathrel{\widehat{=}}$ **extends** M
  state st
  protected sc
  pps
  • act
end

process $P \mathrel{\widehat{=}}$ **extends** Q
  state st
  protected $sc \mathrel{\widehat{=}} [\textbf{super}\ sc]$
  pps
  • act
end

$=$

process $Q \mathrel{\widehat{=}}$ **extends** M
  state st
  protected sc
  pps
  • act
end

process $P \mathrel{\widehat{=}}$ **extends** Q
  state st
  pps
  • act
end

proof

The meaning of P on the left-hand side is defined as:

$$P \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{begin state} \mathrel{\widehat{=}} Q.st \wedge P.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ P.st} \\ \quad Q.pps_2{}^{ref}{}_{\wedge\ \Xi\ P.st} \\ \quad P.pps \\ \quad \textbf{protected}\ P.sc \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!]P.act \\ \textbf{end} \end{array} \right)$$

According to the semantics of **super** clause, $P.sc$ has the same meaning as $Q.sc$; therefore, P becomes:

$$P \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{begin state } \mathrel{\widehat{=}}\ Q.st \land P.st \\ \quad Q.pps_{1\,\land\ \Xi\ P.st} \\ \quad Q.pps_2{}^{ref}{}_{\land\ \Xi\ P.st} \\ \quad P.pps \\ \quad \textbf{protected } Q.sc \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \land\ \ P.st]\!]P.act \\ \textbf{end} \end{array}\right)$$

The above process is the exact meaning of P, in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar and we omit it here.

□

Element Interchanges

A state component st_2 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 P process hierarchy, so we do not need to check, for example, that $st_2 \notin R.st$, since according to our typing rules it must be satisfied by a valid specification in *OhCircus*.The provisos consider $P.st_2$ as a protected element.

If st_2 is a protected state component it can be moved to P if its subprocesses, excepting those that are also subprocesses of R, do not declare a protected state component equals to st_2.

Law 10 (moving state component to subprocess).

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

provided

$(\rightarrow)\ \forall S \mid S \leq P \land \neg\ (S \leq R) \bullet \neg\ occurs(st_2, S.pps) \land \neg\ occurs(st_2, S.act)$

$(\leftarrow)\ \forall S \mid S \leq P \land \neg\ (S \leq R) \bullet st_2 \notin PS(S.st)$

proof

We observe that this law is not a compositional application (to remove and after insert st_2 from P to R) of the Law 4 nor Law 7. Move is an atomic operation, there is no intermediate states.

The meaning of P on the left-hand side is defined as:

$$P \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state } \widehat{=}\;\; Q.st \wedge P.st_1 \wedge P.st_2 \\ \quad Q.pps_1{}^{ref}_{\wedge\ \Xi\ \wedge P.st_1 \wedge P.st_2} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ \wedge P.st_1 \wedge P.st_2} \\ \quad P.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\;\; P.st_1 \wedge P.st_2]\!]P.act \\ \textbf{end} \end{array}\right)$$

= [By A.1]

$$\textbf{var}\ Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\;\; P.st_1 \wedge P.st_2]\!] \quad (12)$$
$$P.act, Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps)$$

= [By A.3]

$$\exists\, Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \wedge Q.st.decl' \wedge P.st_1.decl' \wedge \quad (13)$$
$$P.st_2.decl' \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\;\; P.st_1 \wedge P.st_2]\!]P.act,$$
$$Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps)$$

Considering that $\exists\, x, x', y, y' : T \bullet pred(x) \wedge pred(y) \Rightarrow \exists\, x, x' : T \bullet pred(x)$ and st_2 does not occurs in $P.pps$ nor $P.act$ the above predicate becomes:

$$\exists\, Q.st.decl \wedge P.st_1.decl \wedge Q.st.decl' \wedge P.st_1.decl' \bullet \quad (14)$$
$$ER(Q.act[\![Q.st \mid Q.st \wedge\;\; P.st_1]\!]P.act, Q.st.inv \wedge P.st_1.inv, Q.pps \wedge P.pps)$$

= [By A.3]

$$\textbf{var}\ Q.st.decl \wedge P.st_1.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\;\; P.st_1]\!]P.act, \quad (15)$$
$$Q.st.inv \wedge P.st_1.inv, Q.pps \wedge P.pps)$$

= [By A.1]

$$P \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state } \widehat{=}\;\; Q.st \wedge P.st_1 \\ \quad Q.pps_1 {}_{\wedge\ \Xi\ \wedge P.st_1} \\ \quad Q.pps_2 \overset{ref}{}_{\wedge\ \Xi\ \wedge P.st_1} \\ \quad P.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\;\; P.st_1]\!]P.act \\ \textbf{end} \end{array}\right)$$

The above process is the exact meaning of P, in the UTP, on the right-hand template. The meaning of R on the left-hand side template is:

$$R \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state } \widehat{=}\;\; Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\ \quad Q.pps_1 {}_{\wedge\ \Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\ \quad Q.pps_2 \overset{ref}{}_{\wedge\ \Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\ \quad P.pps_1 {}_{\wedge\ \Xi\ R.st} \\ \quad P.pps_2 \overset{ref}{}_{\Xi\ R.st} \\ \quad R.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\;\; P.st_1 \wedge P.st_2]\!]P.act \\ \qquad [\![Q.st \wedge\;\; P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st]\!] \\ \quad R.act \\ \textbf{end} \end{array}\right)$$

$= [P.st_2 = R.st_2]$

$$R \;\widehat{=}\; \left(\begin{array}{l} \textbf{begin state } \widehat{=}\;\; Q.st \wedge P.st_1 \wedge R.st_2 \wedge R.st \\ \quad Q.pps_1 {}_{\wedge\ \Xi\ P.st_1 \wedge R.st_2 \wedge R.st} \\ \quad Q.pps_2 \overset{ref}{}_{\wedge\ \Xi\ P.st_1 \wedge R.st_2 \wedge R.st} \\ \quad P.pps_1 {}_{\wedge\ \Xi\ R.st} \\ \quad P.pps_2 \overset{ref}{}_{\Xi\ R.st} \\ \quad R.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\;\; P.st_1 \wedge R.st_2]\!]P.act \\ \qquad [\![Q.st \wedge\;\; P.st_1 \wedge R.st_2 \mid Q.st \wedge P.st_1 \wedge R.st_2 \wedge R.st]\!] \\ \quad R.act \\ \textbf{end} \end{array}\right)$$

$= [P.pps \ and \ P.act \ does \ not \ access \ R.st_2]$

$$
R \ \widehat{=} \left(\begin{array}{l}
\textbf{begin state} \ \widehat{=} \ Q.st \wedge P.st_1 \wedge R.st_2 \wedge R.st \\
\quad Q.pps_{1\wedge \ \Xi \ P.st_1 \wedge R.st_2 \wedge R.st} \\
\quad Q.pps_2{}^{ref}{}_{\wedge \ \Xi \ P.st_1 \wedge R.st_2 \wedge R.st} \\
\quad P.pps_{1\wedge \ \Xi \ R.st \wedge R.st_2} \\
\quad P.pps_2{}^{ref}{}_{\Xi \ R.st \wedge R.st_2} \\
\quad R.pps \\
\quad \bullet \ Q.act [\![Q.st \mid Q.st \wedge \ P.st_1]\!] P.act \\
\qquad [\![Q.st \wedge \ P.st_1 \mid Q.st \wedge P.st_1 \wedge R.st_2 \wedge R.st]\!] \\
\quad R.act \\
\textbf{end}
\end{array} \right)
$$

The above process is the exact meaning of R, in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar.

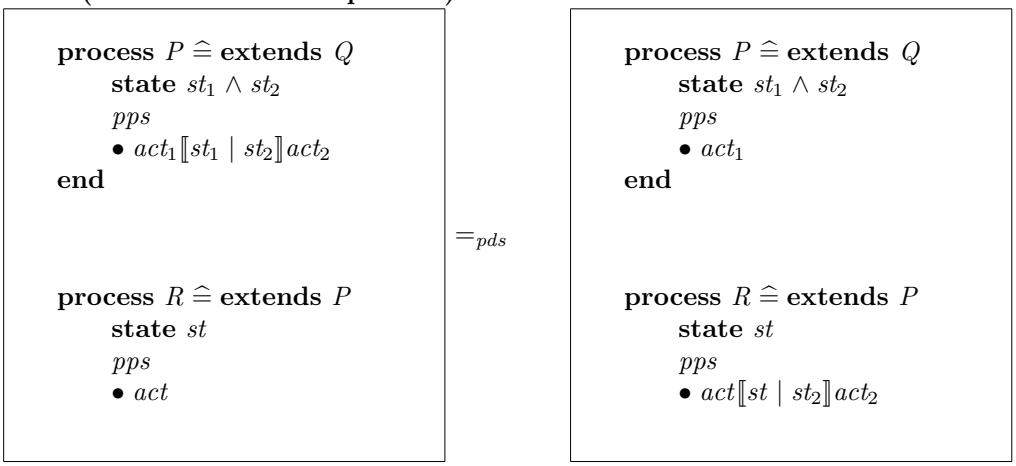
□

If the main action of a process P 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* excepting R and its subprocesses (indirectly). Finally, P cannot be used by any of processes declared in *pds*, excepting by inheritance as already mentioned.

The semantics of R is unchanged as a consequence of the meaning of inheritance. The implicit main action of R, $R.\overline{act}$ is given by its declared main action $R.act$ in parallel composition with the implicit main action of P, $P.\overline{act}$, so $R.\overline{act} = P.\overline{act} [\![P.st \mid R.st]\!] R.act$. Finally, note that act_2 is restricted to change st_2. It cannot know about $R.st$ state components, excepting those that are inherited, including st_2.

The application of this law in the opposite direction will also change the behavior of P, so it cannot be used by any process in *pds* except through inheritance, directly by R, and indirectly by its subprocesses. The semantics of R and its subprocesses is unchanged since act_2 will be part of the implicit main action of R, no matter if it is in P or in R.

Law 11 (move action to subprocess).



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

proof

The meaning of P is changed, otherwise the meaning of R is unchanged just as the meaning of the whole program, since no matter the localization of act_2 if we consider the law provisos.

The meaning of R in the left-hand side template is:

$$
R \mathrel{\widehat{=}} \left(\begin{array}{l}
\textbf{begin state} \mathrel{\widehat{=}} Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\
\quad Q.pps_{1\wedge} \ {}^{\Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad Q.pps_2 \ {}^{ref}{}_{\wedge\ \Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad P.pps_{1\wedge} \ {}^{\Xi\ R.st} \\
\quad P.pps_2 \ {}^{ref}{}_{\Xi\ R.st} \\
\quad R.pps \\
\quad \bullet\ Q.act [\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!] (P.act_1 [\![P.st_1 \mid P.st_2]\!] P.act_2) \\
\qquad [\![Q.st \wedge\ P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st]\!] \\
\quad R.act \\
\textbf{end}
\end{array}\right)
$$

$= [R.act \textit{ is able to access } Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st;$
$\textit{associativity of interleaving operator}]$

$$R \;\widehat{=}\; \left(\begin{array}{l}
\textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\
\quad Q.pps_1 {}^{ref}_{\wedge\ \Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad Q.pps_2 {}^{ref}_{\wedge\ \Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad P.pps_1 {}^{ref}_{\wedge\ \Xi\ R.st} \\
\quad P.pps_2 {}^{ref}_{\Xi\ R.st} \\
\quad R.pps \\
\quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!]P.act_1 \\
\qquad [\![Q.st \wedge\ P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st]\!] \\
\quad R.act[\![R.st \mid P.st_2]\!]P.act_2 \\
\textbf{end}
\end{array}\right)$$

$= [P.act_2 = R.act_2]$

$$R \;\widehat{=}\; \left(\begin{array}{l}
\textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\
\quad Q.pps_1 {}^{ref}_{\wedge\ \Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad Q.pps_2 {}^{ref}_{\wedge\ \Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad P.pps_1 {}^{ref}_{\wedge\ \Xi\ R.st} \\
\quad P.pps_2 {}^{ref}_{\Xi\ R.st} \\
\quad R.pps \\
\quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!]P.act_1 \\
\qquad [\![Q.st \wedge\ P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st]\!] \\
\quad R.act[\![R.st \mid P.st_2]\!]R.act_2 \\
\textbf{end}
\end{array}\right)$$

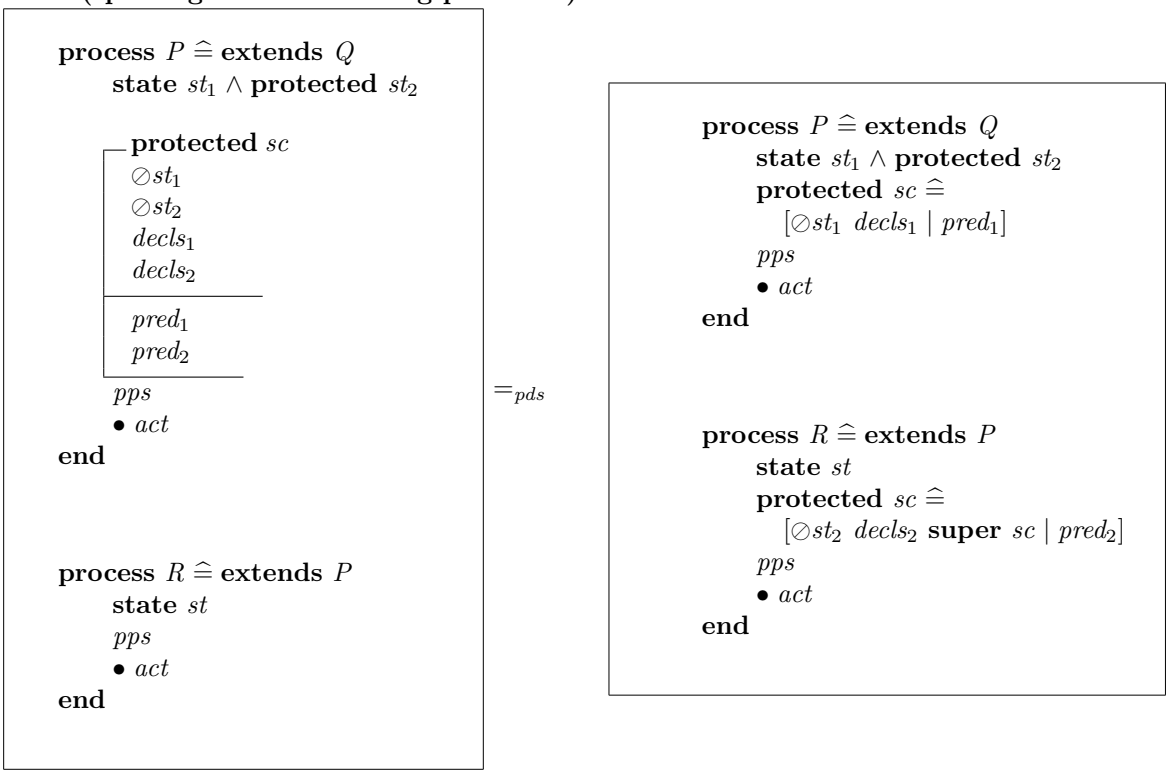
That is the exact meaning of R, in the UTP, on the right-hand template. This concludes our proof. The application of this law in the reverse direction is similar.

□

If part of the behavior 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 its redefinition.

The state components of P 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.

Law 12 (splitting a schema among processes).



provided
$(\leftrightarrow)\ \forall S \mid S \leq P \land \neg\,(S \leq R) \bullet \neg\, occurs(st_2, S.pps) \land \neg\, occurs(st_2, S.act) \land \neg\, impact(st_1, st_2)$ [2]
$(\rightarrow)\ PL(st_2) \land N(sc) \notin N(R.pps)$

proof
The meaning of P on the left-hand side is defined as:

$$P \ \widehat{=}\ \left(\begin{array}{l} \textbf{begin state}\ \widehat{=}\ \ Q.st \land P.st_1 \land P.st_2 \\ \quad Q.pps_1{}_{\land\ \Xi\ \land P.st_1 \land P.st_2} \\ \quad Q.pps_2{}^{ref}_{\land\ \Xi\ \land P.st_1 \land P.st_2} \\ \quad P.pps \\ \quad \textbf{protected}\ P.sc \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \land\ \ P.st_1 \land P.st_2]\!]P.act \\ \textbf{end} \end{array}\right)$$

[2] $impact(st_1, st_2)$ is true iff the value of a state component st_1 is affected by the value of st_2

= [By A.1]

$$\textbf{var}\ Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!]P.act, \quad (16)$$
$$P.act, Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps \wedge P.sc)$$

= [By A.3]

$$\exists\, Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \wedge Q.st.decl' \wedge P.st_1.decl' \wedge \quad (17)$$
$$P.st_2.decl' \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!]P.act,$$
$$Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps \wedge$$
$$sc \mathrel{\widehat{=}} [\oslash st_1\ decls_1 \mid pred_1] \wedge [\oslash st_2\ decls_2 \mid pred_2])$$

Considering that $\exists\, x, x', y, y' : T \bullet pred(x) \wedge pred(y) \Rightarrow \exists\, x, x', y, y' : T \bullet pred(x)$ and the st_1 values are independent of those of st_2 and st_2 does not occur in $P.pps$ nor in $P.act$ we can remove $[\oslash st_2\ decls_2 \mid pred_2]$ from sc.

$$\exists\, Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \wedge Q.st.decl' \wedge P.st_1.decl' \wedge P.st_2.decl' \quad (18)$$
$$\bullet\ ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!]P.act,$$
$$Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps \wedge sc \mathrel{\widehat{=}} [\oslash st_1\ decls_1 \mid pred_1])$$

= [By A.3]

$$\textbf{var}\, Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge \quad (19)$$
$$P.st_1 \wedge P.st_2]\!]P.act, Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv,$$
$$Q.pps \wedge P.pps \wedge sc \mathrel{\widehat{=}} [\oslash st_1\ decls_1 \mid pred_1])$$

= [By A.1]

$$P \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{begin state} \mathrel{\widehat{=}} Q.st \wedge P.st_1 \wedge P.st_2 \\ \quad Q.pps_1 {}^{\wedge\ \Xi\ \wedge P.st_1 \wedge P.st_2} \\ \quad Q.pps_2 {}^{ref}{}^{\wedge\ \Xi\ \wedge P.st_1 \wedge P.st_2} \\ \quad P.pps \\ \quad \textbf{protected}\ sc \mathrel{\widehat{=}} [\oslash st_1\ decls_1 \mid pred_1] \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!]P.act \\ \textbf{end} \end{array}\right)$$

The above process is the exact meaning of P, in the UTP, on the right-hand template. The meaning of R on the left-hand side template is:

$$
R \;\widehat{=}\; \left(\begin{array}{l}
\textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\
\quad Q.pps_1 \wedge_{\Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad Q.pps_2 \overset{ref}{\wedge}_{\Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad P.pps_1 \wedge_{\Xi\ R.st} \\
\quad P.pps_2 \overset{ref}{}_{\Xi\ R.st} \\
\quad \textbf{protected}\ sc \;\widehat{=}\; [\oslash st_1\ decls_1 \mid pred_1] \wedge \\
\qquad [\oslash st_2\ decls_2 \mid pred_2] \\
\quad R.pps \\
\quad \bullet\ Q.act [\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!] P.act \\
\qquad [\![Q.st \wedge\ P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st]\!] \\
\quad R.act \\
\textbf{end}
\end{array}\right)
$$

From the semantics of **super**, the above process is equivalent to:

$$
R \;\widehat{=}\; \left(\begin{array}{l}
\textbf{begin state} \;\widehat{=}\; Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\
\quad Q.pps_1 \wedge_{\Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad Q.pps_2 \overset{ref}{\wedge}_{\Xi\ P.st_1 \wedge P.st_2 \wedge R.st} \\
\quad P.pps_1 \wedge_{\Xi\ R.st} \\
\quad P.pps_2 \overset{ref}{}_{\Xi\ R.st} \\
\quad \textbf{protected}\ sc \;\widehat{=}\; [\textbf{super}\ sc\ \oslash\ st_2\ decls_2 \mid pred_2] \\
\quad R.pps \\
\quad \bullet\ Q.act [\![Q.st \mid Q.st \wedge\ P.st_1 \wedge P.st_2]\!] P.act \\
\qquad [\![Q.st \wedge\ P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st]\!] \\
\quad R.act \\
\textbf{end}
\end{array}\right)
$$

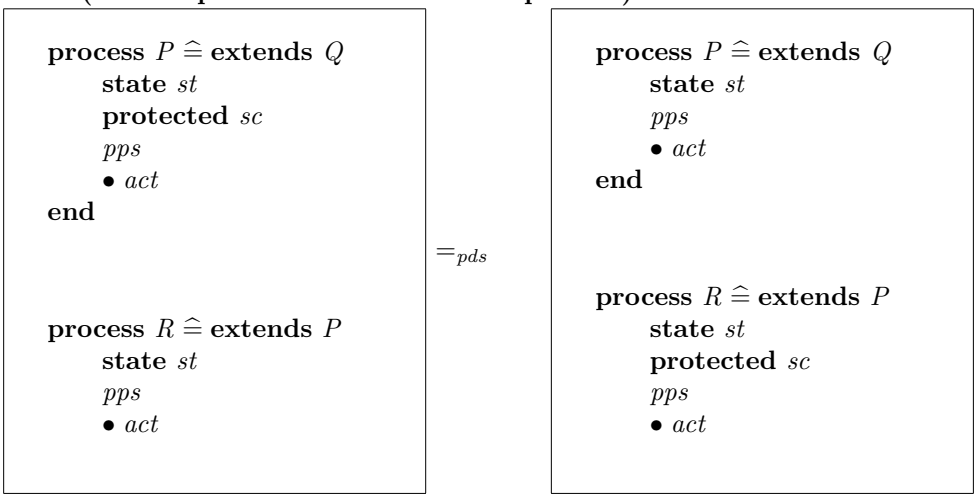
The above process is the exact meaning of R, in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar.

□

To move a schema sc 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, excepting those that are also subprocesses of R. Note that we can apply this law, even if a subprocess of P (excepting R) has a redefinition of sc.

To bring a protected schema $R.sc$ to P, where $R < P$, we must guarantee that P neither its subprocesses, excepting those that are also subprocesses of R, have schema equals to sc.

Law 13 (move a protected schema to subprocess).



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

proof
The meaning of P in the left-hand side is defined as:

$$P \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{begin state } \mathrel{\widehat{=}}\ Q.st \wedge P.st \\ \quad Q.pps_1{}^{ref}_{\wedge\ \Xi\ \wedge P.st} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ \wedge P.st} \\ \quad P.pps \\ \quad \textbf{protected } P.sc \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!]P.act \\ \textbf{end} \end{array}\right)$$

= [By A.1]

$$\textbf{var} Q.st.decl \wedge P.st.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!] \quad (20)$$
$$P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps \wedge P.sc)$$

= [By A.3]

$$\exists\, Q.st.decl \wedge P.st.decl, Q.st.decl' \wedge P.st.decl' \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!] \quad (21)$$
$$P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps \wedge P.sc)$$

Considering an action A, $ER(A, inv, pps \wedge sc) = ER(A, inv, pps)$ if $\neg\ occurs(sc, A)$ and $\neg\ occurs(sc, pps)$. Considering this and the provisos we have that the semantics of P is:

$$\exists\, Q.st.decl \wedge P.st.decl, Q.st.decl' \wedge P.st.decl' \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!] \\ P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \tag{22}$$

= [By A.3]

$$\mathbf{var}\, Q.st.decl \wedge P.st.decl \bullet ER(Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!] \\ P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \tag{23}$$

= [By A.1]

$$P \mathrel{\widehat{=}} \left(\begin{array}{l} \mathbf{begin\ state} \mathrel{\widehat{=}} Q.st \wedge P.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ \wedge P.st} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ \wedge P.st} \\ \quad P.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!] P.act \\ \mathbf{end} \end{array}\right)$$

This is the meaning of P, in the UTP, on the right-hand side template. The meaning of R in the left-hand side template is:

$$R \mathrel{\widehat{=}} \left(\begin{array}{l} \mathbf{begin\ state} \mathrel{\widehat{=}} Q.st \wedge P.st \wedge R.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ P.st \wedge R.st} \\ \quad Q.pps_2{}^{ref}_{\wedge\ \Xi\ P.st \wedge R.st} \\ \quad P.pps_1{}_{\wedge\ \Xi\ R.st} \\ \quad P.pps_2{}^{ref}_{\Xi\ R.st} \\ \quad \mathbf{protected}\ P.sc \\ \quad R.pps \\ \quad \bullet\ Q.act[\![Q.st \mid Q.st \wedge\ P.st]\!] P.act \\ \qquad [\![Q.st \wedge\ P.st \mid Q.st \wedge P.st \wedge R.st]\!] \\ \quad R.act \\ \mathbf{end} \end{array}\right)$$

$= [P.sc = R.sc]$

$$R \mathrel{\widehat{=}} \left(\begin{array}{l} \textbf{begin state } \mathrel{\widehat{=}}\ Q.st \wedge P.st \wedge R.st \\ \quad Q.pps_1{}_{\wedge\ \Xi\ P.st \wedge R.st} \\ \quad Q.pps_2{}^{ref}{}_{\wedge\ \Xi\ P.st \wedge R.st} \\ \quad P.pps_1{}_{\wedge\ \Xi\ R.st} \\ \quad P.pps_2{}^{ref}{}_{\Xi\ R.st} \\ \quad \textbf{protected } R.sc \\ \quad R.pps \\ \quad \bullet\ Q.act [\![Q.st \mid Q.st \wedge\ P.st]\!] P.act \\ \qquad [\![Q.st \wedge\ P.st \mid Q.st \wedge P.st \wedge R.st]\!] \\ \quad R.act \\ \textbf{end} \end{array} \right)$$

The above process is the exact meaning of R, in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar and we omit it here.

□

A default schema sc can be moved from P to R, where $R < P$, if sc is not being used by P and is not defined in R. As sc is default, R is not aware about it, and we must check if R does not define a schema homonymous to sc. It must be clear that the schema $R.sc$, if exists, is not a overriding of $P.sc$ since it is default.

To move the default schema $R.sc$ to P the proviso must guarantee that R doesn't use sc, since R cannot access, directly, a default schema in P. About P, it cannot has a schema equals to sc and neither of its superprocesses define a protected schema homonymous to sc, because it would create an invalid re-definition of this schema. The function PS selects the protected schemas from a set.

Law 14 (move a default schema to subprocess).

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

provided
$(\rightarrow) \neg\ occurs(sc, P.pps) \wedge \neg\ occurs(sc, P.act) \wedge N(sc) \notin N(R.pps)$
$(\leftarrow) \neg\ occurs(sc, R.pps) \wedge \neg\ occurs(sc, R.act) \wedge$
$\forall\ T \mid P < T \bullet N(sc) \notin N(PS(T.pps))$

proof

This proof is similar to Law 13 and we omit it here.

□

Subprocess Extraction
In the initial specification of a system it is common to model processes with a very specific behavior that hides a generic behavior specialized in face of a particular situation. Therefore we develop a composite law that extracts from a process this generic behavior as a superprocess specializing it with a subprocess. This promotes code reuse and favor a better conceptual representation of the system.

Law 15 (subprocess extraction).

This law is composite because its is obtained from the successive application of some of the simple/composite laws presented before.

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
As already mentioned, this law is derived from previous laws. Particularly, it can be proved from : Law 3 (process elimination), Law 11 (move action to subprocess), Law 12 (splitting a schema among processes), Law 13 (move a protected schema to subprocess), Law 14 (move a default schema to subprocess) and Law 10 (moving state component to subprocess).

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

$=$ [*By Law* 3 *and renaming*]

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

process $P \mathrel{\widehat{=}}$ **extends** R
  • *Skip*
end

$=$ [*By Law* 11]

process $R \mathrel{\widehat{=}}$ **extends** Q
  state $st_1 \wedge st_2$
  $pps_{1\wedge} \; \Xi \; {}_{st_2}$
  pps_2
  • act_1
end

process $P \mathrel{\widehat{=}}$ **extends** R
  • act_2
end

The Law 12 (splitting a schema among processes) is then applied for each schema in $P.pps_2$. The set $P.pps_2'$ stands for the schemas in $P.pps_2$ affected by the law and $S.pps_2''$ for those created in S.

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

process $P \mathrel{\widehat{=}}$ **extends** R
  pps_2''
  • act_2
end

The Laws 13 (move a protected schema to subprocess) and 14 (move a default schema to subprocess) can be or not applied at this point, the application of this step is optional and for the sake of conciseness we omit it here.

$=$ [*By Law* 10]

process $R \mathrel{\widehat{=}}$ **extends** Q
  state st_1
  pps_1
  pps_2'
  • act_1
end

process $P \mathrel{\widehat{=}}$ **extends** R
  state st_2
  pps_2''
  • act_2
end

This concludes the derivation.

□

This law can be applied in every context where we have a logical division of schemas and actions concerning state components elements. The unique proviso, in both application directions, must guarantees that R is not used in pds.

4.3 Completeness

The algebraic laws developed in this work focus on the improvement of the process inheritance relations in an *OhCircus* specification. They address state components, schemas and main actions of processes engaged in a hierarchy. Our laws are not concerned with transformations for CSP actions or classes; the works [23] and [3] have already dealt with these topics. Although our work can be part of a completeness theory for *OhCircus*, it is not our intent to establish it, so we just present how process inheritance relations can be removed from specification.

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 subprocess from the target specification; this is exactly what our laws provide if guided by a strategy. In a reduction strategy we apply the laws in a opposite than that applied in the development phase.

Our strategy for subprocess removal can be summarized as the application, in opposite direction, of the laws used in our composite Law 15 (subprocess extraction). Therefore, in an high level view, we must apply the laws: Law 10 (moving state component to subprocess), Law 14 (move a default schema to subprocess), Law 13 (move a protected schema to subprocess), Law 12 (splitting a schema among processes), Law 11 (move action to subprocess) and Law 3 (process elimination) in this order.

Some other simple laws might be necessary in the reduction process. For example, we change the visibility of all schemas and state components of the subprocess S, which we want to move to P (S's superprocess), to **protected**. This transformation might generate name conflicts that can be solved with a simple renaming. After this, Law 10 (moving state component to subprocess) can be successively applied to move up each state component from S to P.

5 Case Study

Consider the process *Buffer* as defined in Figures 5 and 6. Our intention is to transform this design into a more reusable one, as presented in Section 2.1 (see Figures 2, 3 and 4). The process *Buffer* (in Figures 5 and 6) encompasses two abstractions: an 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.

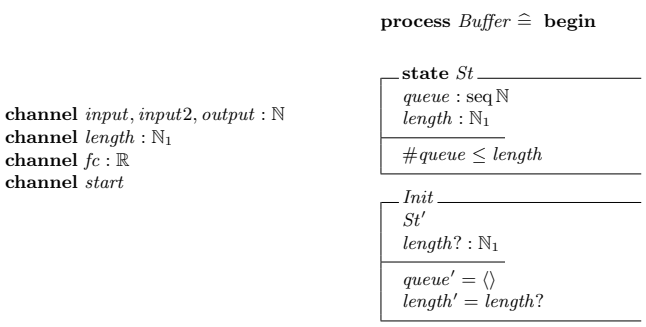


Fig. 5. *Buffer* without inheritance - Part I

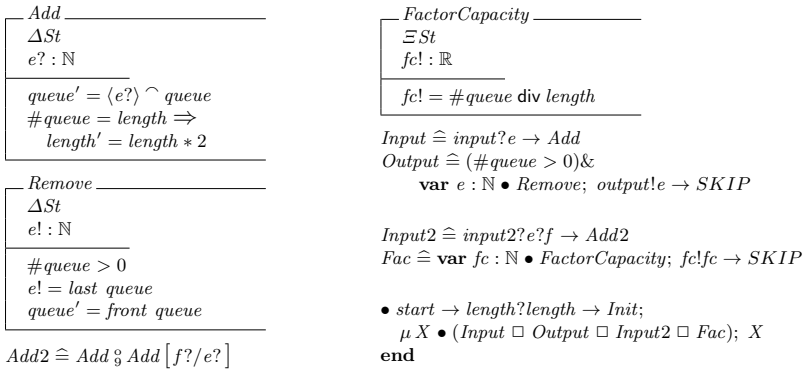


Fig. 6. *Buffer* without inheritance - Part II

Separating these concerns increases reuse and maintainability, and the nature of the design is more faithfully reflected. To achieve these benefits Law 15 (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 15.

Figure 7 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 [7]. With these transformations we can apply Law 15. As explained in the previous section, it embodies several small transformations: Law 3 is applied to create the subprocess *BufferImp*; Law 13 is applied to the schema *Add*; the state component St_2, its related initial clause and main action are moved to *BufferImp*, by the Laws 10 and 14 and 11, respectively; and the state St_1, its related initial clause and main action remains in *Buffer*. Several rename operations, as St_1 to

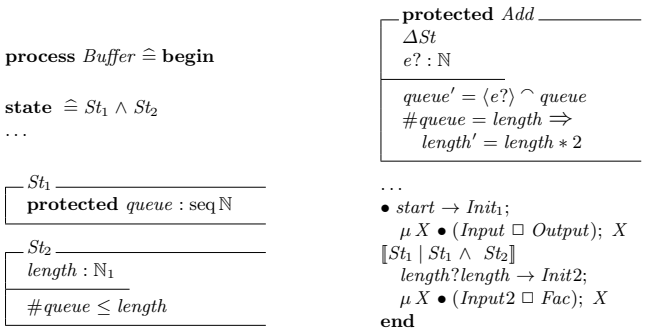


Fig. 7. *Buffer* adaptations

St and $Init_1$ to *Init* are omitted for the sake of conciseness. The result is the design in Section 2.1.

6 Tool support

We developed a tool to support our strategy based on the Eclipse Modelling Framework (EMF), 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.

Using Xtext we are able to generate an Ecore metamodel from an EBNF-grammar like description of *OhCircus*. The next step was the encoding of the algebraic laws, using ATL, which provides reasonable support for model-to-model transformation. For each algebraic law two ATL modules were created, one for each direction of application.

The only drawback of using ATL is that it is normally used in the context of transformations involving distinct source and target models. In our case, the proposed laws relate elements of a same language (*OhCircus*). In this case, one has to encode auxiliary rules to capture the fact that some elements are preserved by the transformation, by explicitly copying them from the source to the destination of the transformation. Also, when an element is removed as an effect of the transformation, such copying rules have to be disabled, not only for the element itself, but for all its constituent elements.

After having implemented all ATL modules, we built a simple tool in Java to mechanically apply the laws. The Java class *ATLTransformation* acts like an ATL engine. It receives the input models, the input metamodel, the output metamodel and the ATL module. With all this information provided, it executes the transformation generating the target model.

Currently we have developed a subset of the laws as a proof of concept [3].

7 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 [29] for CSP, based on the failures model [12], 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 signalize the superprocess elements which will be visible to its subprocesses. We needed to introduce a new parallel operator, and its UTP semantics, to be able to define the meaning of process inheritance.

Several works have addressed the notion of behavioural subtyping [19,14,10,29] [1,5,4,2]. In [1,14] 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 some process algebra model, like failures and failures-divergences proposed for CSP, relating process refinement with inheritance [29].

In [14] a subtype is allowed to extend the behaviour of its supertype, adding new methods, provided there exists a function that maps these new methods as a combination of the supertype methods; this is not allowed in [29]. Here we allow, in a subtype, new methods like in [14] and even more: new state components, method overriding, reducing the non-determinism, and methods that change both inherited and declared state components.

Previous works have proposed refinements and algebraic laws for *Circus* [24,7] and these are consequently applicable to *OhCircus*. In [24] the meaning of refinement of processes and their actions are defined based on forward simulation. It also proposes laws in the process grain, as splitting and indexing processes, as part of a general method of development for *Circus*, based on refinement. The work reported in [24] also includes laws for actions, and provides an accompanying iterative development strategy, involving the application of simulation, action and, most importantly, process refinement. In this context, our work complements [24] 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 [18]. The extension of this work for *OhCircus*, in the form proposed here, is our next immediate goal.

[3] `http://www.cin.ufpe.br/~prga2/TG/CircusRefiningTool/`

A UTP framework

Semantics of processes, schemas and commands developed in [17].

A.1 *OhCircus* Processes

Derived from [17], since the action in the command is submitted to the ER function.

$$\textbf{begin state}\ [decl \mid pred]PPars \bullet A\ \textbf{end} \ \widehat{=}\ \textbf{var}\ decl \bullet ER(A, pred, PPars)$$

$$ER : \mathsf{Action} \nrightarrow \mathbb{P}\,\mathsf{Constraint} \nrightarrow \mathbb{P}\,\mathsf{PParagraph} \nrightarrow \mathsf{Action}$$
$$ER\ a\ \emptyset\ \emptyset = a$$
$$ER\ a\ \emptyset\ p = R(a, p)$$
$$ER\ a\ inv\ \emptyset = E(a, inv)$$
$$ER\ a\ i : invs\ p : pps = ER(E(R(a, E(p, i)), i), invs, pps)$$

$$E : \mathsf{Action} \nrightarrow \mathsf{Constraint} \nrightarrow \mathsf{Action}$$
$$E\ a\ inv = a;\ \textbf{if}\, inv \rightarrow Skip [\!] Stop$$
$$E\ (a\ op\ b)\ inv = E(a, inv)\ op\ E(b, inv)$$

$$E : \mathsf{Paragraph} \nrightarrow \mathsf{Constraint} \nrightarrow \mathsf{Paragraph}$$
$$E\ [udecl;\ ddecl' \mid pred]\ inv = [udecl;\ ddecl' \mid pred \wedge inv]$$

$$R : \mathsf{Action} \nrightarrow \mathsf{Paragraph} \nrightarrow \mathsf{Action}$$
$$R\ a\ sc \mathrel{\widehat{=}} [udecl;\ ddecl' \mid pred] = \ \textbf{if}\ a = sc\ \textbf{then}\ [udecl;\ ddecl' \mid pred]\ \textbf{else}\ a$$
$$R\ (a\ op\ b)\ sc = R(a, sc)\ op\ R(b, sc)$$

A.2 Schema Expressions

$$[udecl;\ ddecl' \mid pred] \ \widehat{=}\ \ ddecl : [\exists\, ddecl' \bullet pred, pred]$$

A.3 Command

$$\textbf{var}\ x : T \bullet A = \exists\ x;\ x' : T \bullet A$$

B Verification of case study in FDR

This Appendix explaining how components of the study cases might be verified in the model checker FDR (for Failure-Divergence-Refinement). The appendix is not intended to detail the study cases or the strategy presented in this work, but to focus on the aspects that ease their verification.

```
-- channel declarations --
n = 2

channel input,Ninput, output : {0..n}
channel input2 : {0..n}.{0..n}
channel div
channel fc:{0..n}

-- auxiliary function --
reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>

-- a regular buffer ---
Buffer(queue) =
#queue < 4 & input?x -> Buffer(<x>^queue)
[]
#queue > 0 & output.head(reverse(queue)) ->
        Buffer(reverse(tail(reverse(queue))))

-- a buffer implementation --

BufferImp(queue,len) =
     (  #queue < 4 & input?x -> Buffer(<x>^queue)
[]
#queue > 0 & output.head(reverse(queue)) ->
            Buffer(reverse(tail(reverse(queue))))
[]
#queue == len and len<=2 & BufferImp(queue,len*2)
        []
        fc.(len - #queue) -> Buffer(queue)
      )
       |||
      (  #queue < (len-1) & input2?x?y ->
            BufferImp(<y>^<x>^queue,len) )

-- verifying subtype without sharing --

Tester2(queue,len) = BufferImp(queue,len) \ {| input2,fc |}

assert Buffer(<>) [F= Tester2(<>,1)
```

```
-- verifying subtyping with sharing

New2Old = All
|~| Old

All = input2?x?y -> P1(x,y)
        [] fc?x -> P3(x)
[] input?x -> New2Old
[] output?x -> New2Old

P1(x,y) = div -> DIV
[] Ninput.x -> P2(y)

P2(y) = div -> DIV
[] Ninput.y -> New2Old

P3(x) = div -> DIV
[] New2Old

DIV = DIV

Old = input?x -> New2Old
[] output?x -> New2Old

Tester(queue,len) =
( (BufferImp(queue,len)
[| {| input, output, input2,fc |} |]
New2Old)
\ {| input2, div,fc |})
[[ Ninput <- input ]]

Tester2(queue,len) = BufferImp(queue,len) \ {| input2,fc |}

assert Buffer(<>) [F= Tester(<>,1)
```

C Access Levels Equivalence

Some laws change the access level of schemas or state components, the proofs of these laws use the bellow lemmas based on the similar lemmas for ROOL[9]:

Lemma 1 (access levels equivalence between schema in semantics level).

Consider Γ and Γ' proper typing environments, where $\Gamma.vis\ P\ sc = default$ and $\Gamma'.vis\ P = \Gamma.vis\ P \oplus \{sc \mapsto protected\}$. We have that:
$\Gamma, pds \rhd P : \mathbf{Process} = \Gamma', pds \rhd P : \mathbf{Process}$

provided
$(\rightarrow)$ $(1) \forall\, Q \mid P < Q \bullet (\forall\, s \in Q \mid N(s) = N(sc) \rightarrow\ \neg\, PL(s))$
$(2) \forall\, R \mid R < P \bullet \neg\ occurs(sc, R.act) \wedge \forall\, s \in R.pps \bullet (\neg\ N(sc) = N(s) \wedge \neg\ occurs(sc, s))$
$(\leftarrow)$ $(1) \forall\, Q \mid P < Q \bullet sc \notin P.pps$ $(2) \forall\, R \mid R < P \bullet sc \notin R.pps$

proof By induction, since the semantics is defined in terms of the extended typing system which does not enforce the visibility constraints, the difference between Γ and Γ' is irrelevant.

Lemma 2 (access levels equivalence between state components in semantics level).

Consider Γ and Γ' proper typing environments, where $\Gamma.vis\ P\ st_1 = default$ and $\Gamma'.vis\ P = \Gamma.vis\ P \oplus \{st_1 \mapsto protected\}$. We have that:
$\Gamma, pds \rhd P : \mathbf{Process} = \Gamma', pds \rhd P : \mathbf{Process}$

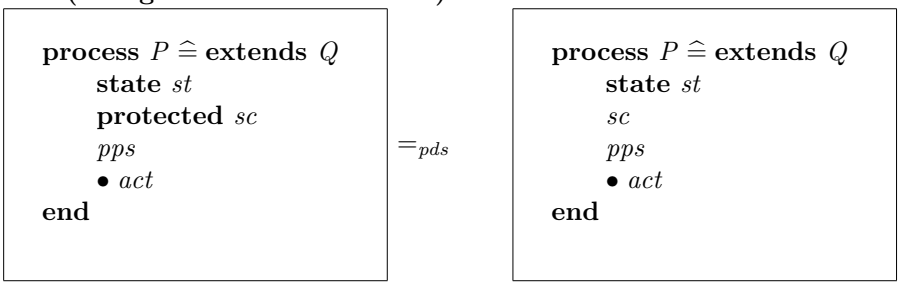
provided
$(\rightarrow)$ $\forall\, R \mid R < P \bullet \neg\ occurs(R.st_1, R.act) \wedge \neg\ occurs(R.st_1, R.pps)$
$(\leftarrow)$ $\forall\, R \mid R < P \bullet st_1 \notin PS(R.st)\ \wedge\ \forall\, Q \mid P < Q \bullet st_1 \notin PS(Q.st)$

proof Similar to Lemma 1.

D Algebraic Laws

D.1 Access Modifications

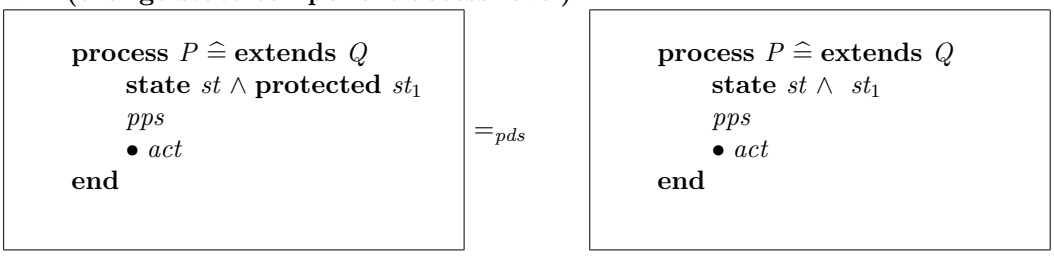
Law 1 (change schema access level).



provided
$(\rightarrow)$ $(1) \forall\, Q \mid P < Q \bullet (\forall\, s \in Q.pps \mid N(s) = N(sc) \rightarrow \neg\, PL(s))$
$(2) \forall\, R \mid R < P \bullet \neg\, occurs(sc, R.act) \wedge \forall\, s \in R.pps \bullet (\neg\, N(sc) = N(s) \wedge \neg\, occurs(sc, s))$
$(\leftarrow)$ $(1) \forall\, Q \mid P < Q \bullet sc \notin P.pps$ $(2) \forall\, R \mid R < P \bullet sc \notin R.pps$

□

Law 2 (change state component access level).



provided
$(\rightarrow)$ $\forall\, R \mid R < P \bullet \neg\, occurs(R.st_1, R.act) \wedge \neg\, occurs(R.st_1, R.pps)$
$(\leftarrow)$ $\forall\, R \mid R < P \bullet st_1 \notin PS(R.st) \ \wedge \ \forall\, Q \mid P < Q \bullet st_1 \notin PS(Q.st)$ □

D.2 Localized Eliminations

Law 3 (process elimination).
$pds\ pd_1 = pds$

where
$pd_1 =$ **process** $P \mathrel{\widehat{=}}$ **begin** $\bullet$ *Skip* **end**

provided
$(\leftrightarrow)\ \neg\ occurs(P, pds)$

□

Law 4 (default schema elimination).

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

$=_{pds}$

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

provided
$(\rightarrow)\ \neg\ occurs(P.sc, P.act) \wedge \neg\ occurs(P.sc, P.pps)$
$(\leftarrow)\ sc \notin P.pps \wedge (\forall\ Q \mid P < Q \bullet sc \notin Q.pps, \mathbf{if} PL(sc))$ □

Law 5 (default state component elimination).

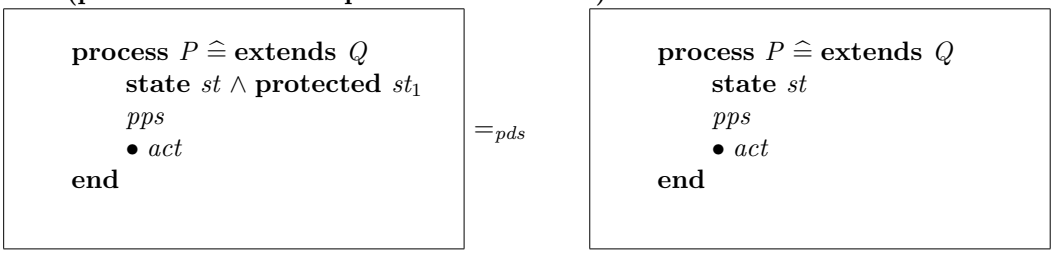
process $P \mathrel{\widehat{=}}$ **extends** Q
state $st \wedge st_1$
pps
$\bullet\ act$
end

$=_{pds}$

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

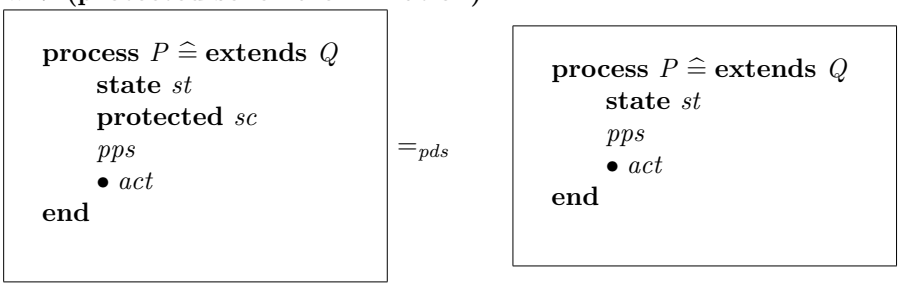
provided
$(\rightarrow)\ \neg\ occurs(P.st_1, P.act) \wedge \neg\ occurs(P.st_1, P.pps)$
$(\leftarrow)\ st_1 \notin P.st$ □

Law 6 (protected state component elimination).



provided
$(\rightarrow)\ \forall R \mid R \leq P \bullet \neg\ occurs(R.st_1, R.act) \wedge \neg\ occurs(R.st_1, R.pps)$
$(\leftarrow)\ \forall R \mid R \leq P \bullet st_1 \notin PS(R.st)\ \wedge\ \forall Q \mid P < Q \bullet st_1 \notin PS(Q.st)$ □

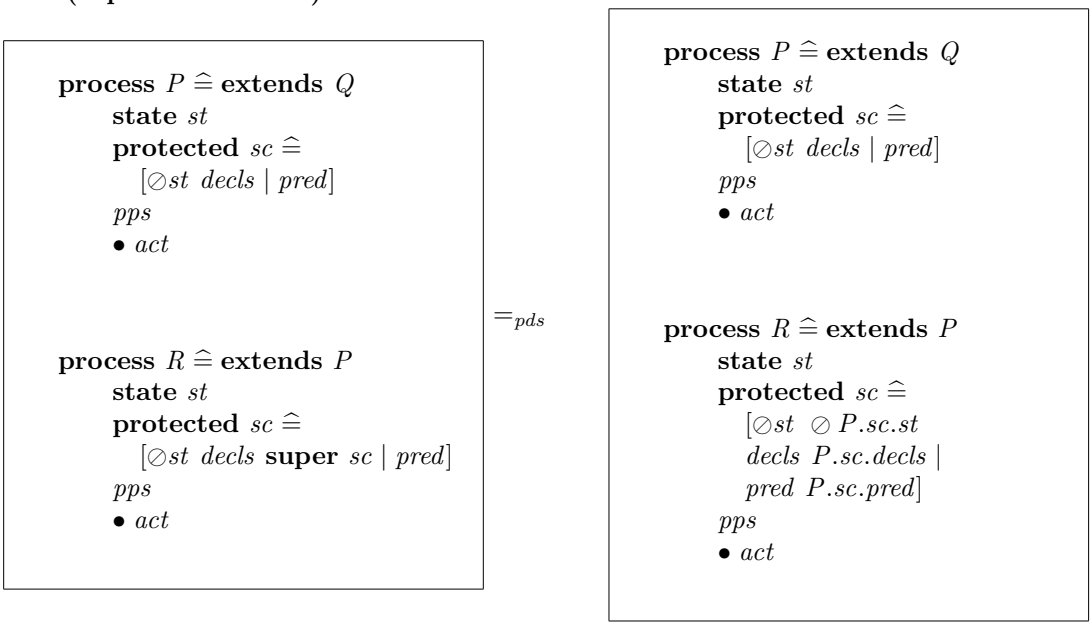
Law 7 (protected schema elimination).



provided
$(\rightarrow)\ \forall R \mid R <\ P \bullet \neg\ occurs(sc, R.act) \wedge \neg\ occurs(sc, R.pps) \vee$
$N(sc) \in N(R.pps) \Rightarrow P.sc \sqsubseteq R.sc \wedge \neg\ occurs(sc, R.sc)$
$(\leftarrow)\ sc \notin P.pps \wedge (\forall R \mid R < P \bullet sc \notin R.pps \vee (sc \in R.pps \wedge P.sc \sqsubseteq R.sc)) \wedge$
$(\forall Q \mid P < Q \bullet sc \notin Q.pps \vee (sc \in Q.pps \wedge Q.sc \sqsubseteq P.sc))$

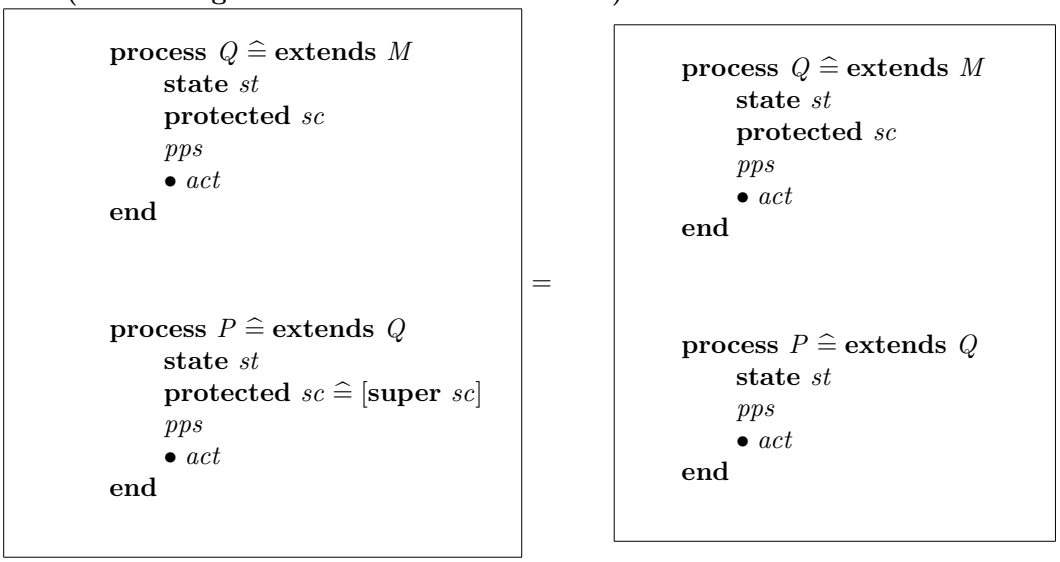
□

Law 8 (super elimination).



□

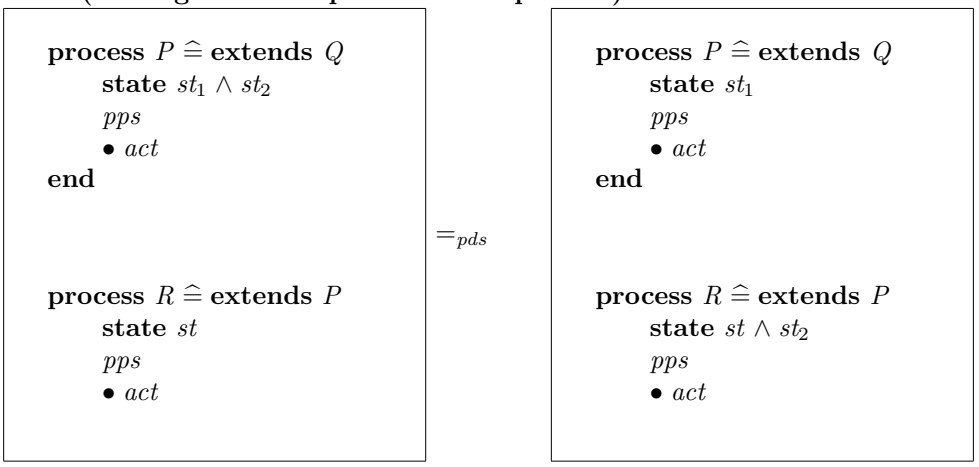
Law 9 (eliminating a trivial schema redefinition).



□

D.3 Element Interchanges

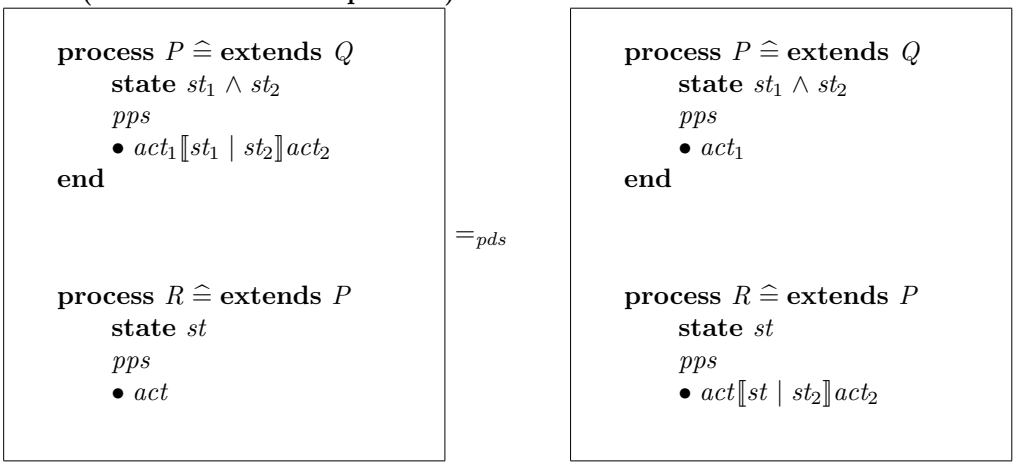
Law 10 (moving state component to subprocess).



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

□

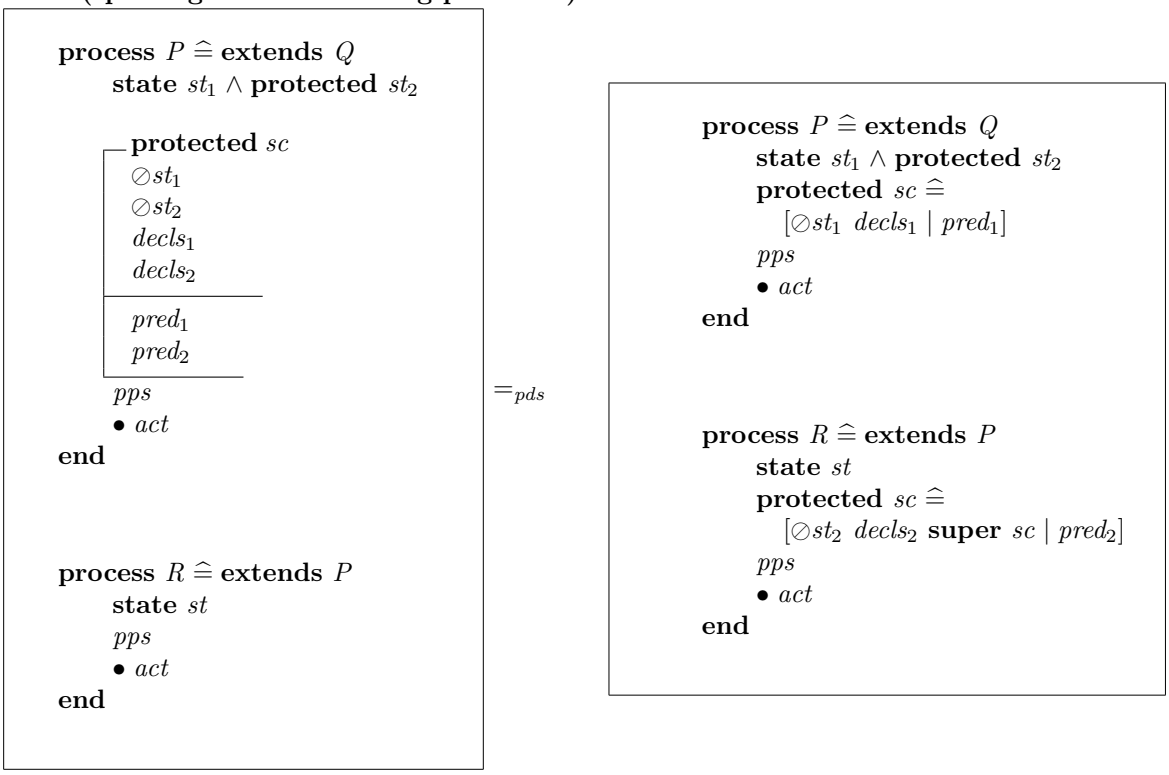
Law 11 (move action to subprocess).



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

□

Law 12 (splitting a schema among processes).

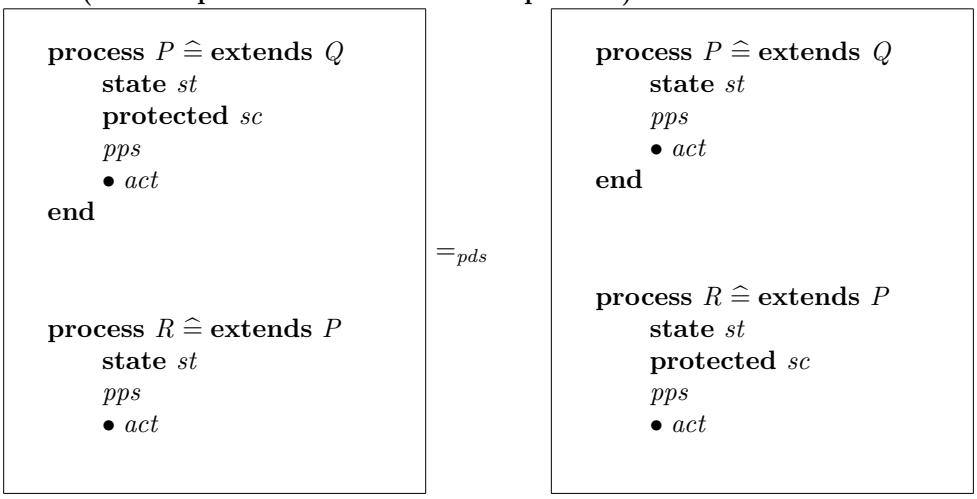


provided
$(\leftrightarrow)$ $\forall S \mid S \leq P \land \neg (S \leq R) \bullet \neg\, occurs(st_2, S.pps) \land \neg\, occurs(st_2, S.act) \land \neg\, impact(st_1, st_2)$ [4]
$(\rightarrow)$ $PL(st_2) \land N(sc) \notin N(R.pps)$

□

[4] $impact(st_1, st_2)$ is true iff the value of a state component st_1 is affected by the value of st_2

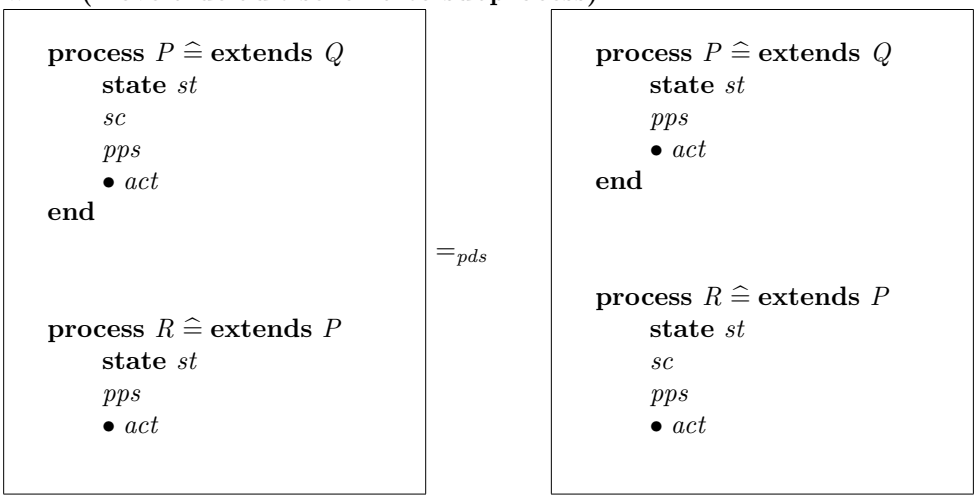
Law 13 (move a protected schema to subprocess).



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

□

Law 14 (move a default schema to subprocess).

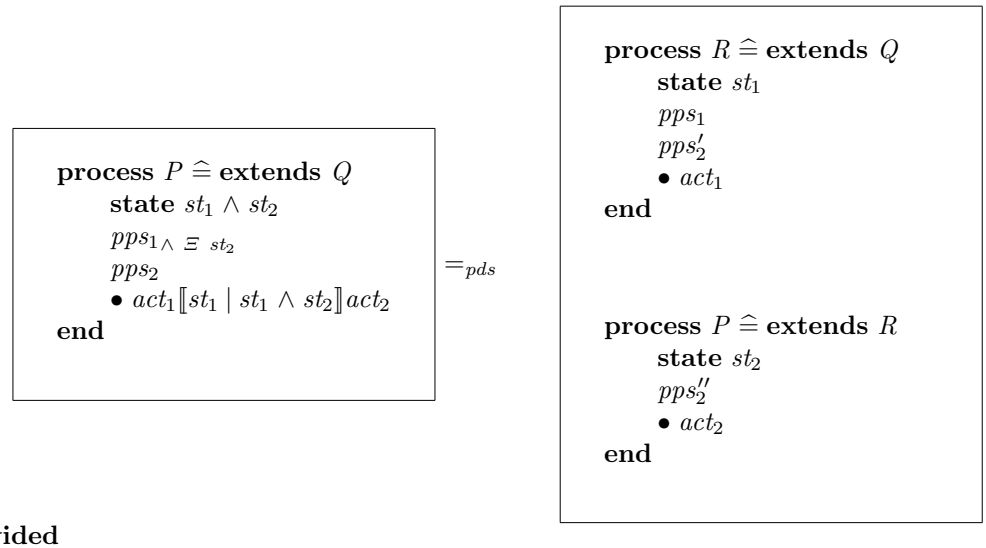


provided
$(\rightarrow)\neg\ occurs(sc, P.pps) \wedge \neg\ occurs(sc, P.act) \wedge N(sc) \notin N(R.pps)$
$(\leftarrow)\neg\ occurs(sc, R.pps) \wedge \neg\ occurs(sc, R.act) \wedge$
$\forall T \mid P < T \bullet N(sc) \notin N(PS(T.pps))$

□

D.4 Subprocess Extraction

Law 15 (subprocess extraction).



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

□

References

1. Pierre America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, London, UK, 1991. Springer-Verlag.
2. C. Balzarotti, Fiorella de Cindio, and Lucia Pomello. Observation equivalences for the semantics of inheritance. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 455–, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.
3. Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. *Sci. Comput. Program.*, 52:53–100, August 2004.
4. H. Bowman, C. Briscoe-smith, J. Derrick, and B. Strulo. On Behavioural Subtyping in LOTOS, 1996.
5. Howard Bowman and John Derrick. A Junction between State Based and Behavioural Specification (Invited Talk). pages 213–239, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.
6. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of Actions in *Circus*. In *Proceedings of REFINE'2002*, Eletronic Notes in Theoretical Computer Science, 2002. Invited paper.
7. 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.
8. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277 – 296, 2005.
9. M. L. Cornlio. *Refactorings as Formal Refinements*. PhD thesis, Centro de Informtica - UFPE, Recife-Brazil, 2004. BC2004-481.
10. Elspeth Cusack. Refinement, conformance and inheritance. *Formal Aspects of Computing*, 3:129–141, 1991. 10.1007/BF01898400.
11. Clemens Fischer. CSP-OZ: a combination of object-Z and CSP. In *Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, pages 423–438, London, UK, UK, 1997. Chapman & Hall, Ltd.
12. C. A. R. Hoare. Communicating Sequential Processes. volume 21, pages 666–677, New York, NY, USA, August 1978. ACM.
13. C.A.R. Hoare and J. He. *Unifying theories of programming*, volume 14. Prentice Hall, 1998.
14. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
15. Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
16. Ernst-Rüdiger Olderog and Heike Wehrheim. Specification and (property) inheritance in CSP-OZ. *Sci. Comput. Program.*, 55(1-3):227–257, March 2005.
17. M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02.
18. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, **21**(1):3 – 32, 2007. The original publication is available at www.springerlink.com.

19. Franz Puntigam. Types for Active Objects Based on Trace Semantics. In *Proceedings FMOODS 96*, pages 4–19. Chapman and Hall, 1996.
20. A. W. Roscoe. Csp and determinism in security modelling. In *In Proc. IEEE Symposium on Security and Privacy*, pages 114–127. Society Press, 1995.
21. A. W. Roscoe. The pursuit of buffer tolerance, 2005.
22. A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
23. 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.
24. Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in *Circus*. *Lecture Notes in Computer Science*, 2391:451–470, 2002.
25. Steve Schneider and Helen Treharne. Communicating B Machines. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 416–435, London, UK, UK, 2002. Springer-Verlag.
26. J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
27. Hans Toetenel and Jan van Katwijk. Stepwise development of model-oriented real-time specifications from action/event models. In *Proceedings of the Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, London, UK, UK, 1991. Springer-Verlag.
28. Peter Wegner and Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '88, pages 55–77, London, UK, 1988. Springer-Verlag.
29. Heike Wehrheim. Behavioral Subtyping Relations for Active Objects. *Form. Methods Syst. Des.*, 23(2):143–170, 2003.
30. Jim Woodcock and Ana Cavalcanti. The Semantics of Circus. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 184–203, London, UK, UK, 2002. Springer-Verlag.
31. Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.