Programming Research Group

GENERIC COMPOSITIONS

Chen, Yifeng

PRG-RR-01-04

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

This paper presents a technique called *generic composition* to provide a neat basis for sequential composition, different kinds of parallel compositions and various higher-order healthiness conditions appearing in a variety of semantic theories. The weak inverse of generic composition is defined. A completeness theorem shows that any predicate can be written in terms of generic composition and its weak inverse, and a number of algebraic laws are given to support reasoning.

1 Introduction

The modeling of programs as predicates coupled with the view of observation-oriented semantics simplifies the semantic interface, benefits from the expressive power of set theory and logic, and allows refinement to be expressed as implication. The use of predicate calculus is especially useful since variable hiding becomes as simple as existential quantification applied to a predicate's corresponding input and output variables.

Such predicative semantics corresponds directly to relational semantics [11, 12, 16] and has been shown to span sequential (imperative, logic and functional) and parallel (synchronous and asynchronous) programming paradigms. This paper helps unify those various applications of the theory by proposing a form of composition which unifies the forms of composition basic to those various paradigms.

In [16] a semantic space is a set of predicates satisfying some *healthiness conditions*. The semantic space PRED without any healthiness condition is a complete lattice, which contains all predicates. Its top and bottom are predicate constants 'false' and 'true' respectively. Its lub (least upper bound) and glb (greatest lower bound) are conjunction and disjunction respectively. The ordering relation is refinement ordering $\sqsubseteq$, such that $A \sqsubseteq B$ if and only if $B \Rightarrow A$ is always true.

We can draw an analog between healthiness conditions and mathematical constraints. For example, condition $x^2 + y^2 \leqslant 1$ specifies an area in a 2D space $\mathbb{R}^2$. The area can be considered the target semantic space and each point is analogous to a predicate. The mathematical condition plays a role similar to a healthiness condition in predicative semantics.

The notion of 'healthiness condition' was first introduced by Dijkstra in [9] for predicate-transformer semantics. In [16] a healthiness condition is written $h(\cdot)$ where h is a higher-order predicate. A predicate A is 'healthy' with respect to this healthiness condition, if $h(A)$ holds. A semantic space may satisfy several healthiness conditions.

A healthiness condition $h(A)$ is *algebraic* if it can be expressed as an equation $A = f(A)$ where f is a *predicate function*. If f is monotonic as well as idempotent under composition, it is called a *healthiness function*.

If all healthiness conditions are the fixed points of some healthiness functions and their composition is idempotent, the composition is then called a *grand healthiness function*.

Writing healthiness conditions in such a restricted way has following advantages:

1. a healthiness function characterizes a link between semantic spaces by transforming each unhealthy predicate to a healthy one in the more detailed space;

2. a healthiness function maps a complete-lattice semantic space to a subset complete-lattice space [16] (page 91);

3. a composition (e.g. sequential composition) is closed in a semantic space iff the composition is closed in regard to the grand healthiness function;

4. according to a major theorem in [16] (page 92), a healthiness function transforms the fixed point of a recursion program to that in a more detailed space if it satisfies a couple of simple conditions;

5. the laws of a basic model can be inherited by a more detailed model if the commands and the grand healthiness function satisfy some simple conditions of idempotence and commutativity (see Chapter 2).

These advantages suggest that we should write healthiness conditions in this form whenever possible.

One might doubt whether this approach is general enough to cope with various healthiness conditions from very diversified semantic models. This is a reasonable concern, especially when those *higher-order healthiness conditions* are not as simple as *conjunctive healthiness conditions* $A = A \wedge R$ or *disjunctive healthiness conditions* $A = A \vee R$. Suppose we have m healthiness conditions (in CSP [15] $m = 8$ and in BSP [23] $m > 10$) and n commands (in CSP $n = 8$, in BSP $n = 13$); then $m \times m$ proofs are needed for the idempotence and commutativity of the healthiness functions and $m \times n$ are needed for the closure of commands. Many technical proofs are not trivial. Furthermore a semantic space may not even be a complete lattice. For example, finite non-determinism is not a complete lattice and hence cannot be represented as the fixed point of a healthiness function.

In order to simplify the semantic studies on healthiness conditions, we propose a new unifying technique called *generic composition*. By writing higher-order healthiness functions with generic compositions, we can reduce the proofs of idempotence and commutativity to some easy-to-check conditions. Although some semantic spaces are not complete lattices, we find it convenient to add either a top or a bottom to the space to 'forge' a complete lattice (see Section 5).

The weak inverse of generic composition is also defined. A completeness theorem shows that any predicate can be written in terms of generic composition and its weak inverse, thus showing the technique has universal expressibility.

By using generic compositions, we have systematically defined (in terms of monotonic and idempotent fixed-point functions) higher-order healthiness conditions from a variety of models, such as Unifying Theories of Programming [16], CSP [13], CCS [17], Game Semantics [1], Z [24], probabilistic programming [10], safety and liveness properties [2] and BSP [23, 20]. The technique has significantly simplified semantic studies of BSP as we shall see in Chapter 2 and 3.

[16] introduces a kind of parallel composition called *parallel-by-merge*. Its definition is based on sequential semantics and therefore compatible with sequential composition. Two programs composed by a parallel composition are first simulated separately and

their disjoint results are exported to a *merge* operation following them. However, as [20] noticed, the merge of BSP's parallel composition may not be a valid merge (see [4]). Parallel-by-merge allows no interference of inputs in that its initial state is copied by both components. A Petri-net [22] counterexample in Section 5.1 shows that there can be 'conflict' between the inputs of the two components in a parallel composition.

We propose a new kind of parallel composition called *parallel-via-medium composition* based on generic composition. The composition is less restricted than parallel-by-merge in the following aspects (cf. [16], Chapter 7):

1. it does not impose any healthiness condition on the medium between the two components in a parallel composition, although the parallel composition has to be closed under the semantic space;

2. the interface between the two components can be explicitly set and any variable can be included in the interface.

The two generalizations fix BSP's parallel composition [20] and lead to a more general parallel composition and simpler reasoning for it.

Section 2 reviews the formalism from [16] required here. Section 3 introduces generic composition and its weak inverse. Section 4 introduces generic-composition-based sequential and parallel compositions and higher-order healthiness conditions. More advanced forms of generic compositions are also introduced in this section. Section 5 shows how generic composition can be used to define parallel compositions and higher-order healthiness conditions for various semantic models.

2 Predicate semantics

We need some conventions for the variables in a predicate. For example if $v \mathrel{\widehat{=}} x, y, z$ is a list of variables, then $\{v\} = \{x, y, x\}$ is its corresponding set of variables from the list.

Symbols	Type of Each Symbol
$x, y, z, r, s, t, x_0, y_0, \cdots$	individual variable
$u, v, w, u_0, v_0, \cdots$	finite list of variables
$\alpha, \beta, \gamma, \alpha_0, \beta_0, \cdots$	set of variables
$\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}, \boldsymbol{\alpha}_0, \boldsymbol{\beta}_0, \cdots$	list of variables in some proposed order

We can decorate a variable x in a number of ways: $\overline{x}$, x', $x.c$, $x.0$, x_1, $x.c.tr$ etc. Decorating a set of variables is the same as decorating each of them. Decorating a list of variables preserves the order of the list. For example, if $v \mathrel{\widehat{=}} x, y, z$ then $v' \mathrel{\widehat{=}} x', y', z'$. A single decoration can be done only once. For example $(v')'$ is not allowed, nor is v'', which is considered the same as $(v')'$. Decoration is useful in expressing different observable aspects of a program variable. For example x and x' can be used to denote the initial and final states of x, respectively.

Using a proposed order for variable lists instead of variable sets is convenient for substitution. For example, $A[\boldsymbol{\beta} := \boldsymbol{\beta}_0]$ modifies A by substituting each variable x in β

with variable x_0. Adding a decoration to $\boldsymbol{\beta}$ creates a list of new variables in the same proposed order. For example, list $\boldsymbol{\beta}_0$ has the same proposed order as $\boldsymbol{\beta}$.

We now review some formalism of [16] required in this paper. The fundamental concept in [16] is that of a *binary relation*. A binary relation is a pair (α, P) where α is a set of variables called the *alphabet* and P is a predicate containing no free variables other than those in α. We abbreviate that pair to P if omitting its alphabet does not lead to confusion.

The alphabet α of a binary relation has two parts α_{in} and α'_{out} satisfying $\alpha = \alpha_{in} \cup \alpha'_{out}$. α_{in} is a set of undashed variables called the *input alphabet*, while α'_{out} is a set of dashed variables called the *output alphabet*.

For example, if x and y are the only program variables, the semantics of program $x := x + 1$ becomes a relation

$$(\{x, y, x', y'\},\ x' = x + 1 \wedge y' = y).$$

The refinement order between two relations A and B with alphabet α is defined by $A \sqsubseteq B \ \widehat{=}\ \forall \boldsymbol{\alpha} \cdot (B \Rightarrow A)$ where $\boldsymbol{\alpha}$ is a list of all variables from α in a proposed order. Sequential composition between two relations is then defined by

$$A\,;\ B \ \widehat{=}\ \exists \boldsymbol{\beta_0} \cdot A[\boldsymbol{\beta'} := \boldsymbol{\beta}_0] \wedge B[\boldsymbol{\beta} := \boldsymbol{\beta_0}]$$

where $\boldsymbol{\beta}_0$ is a list of *new* variables (otherwise we use $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, etc.).

Two special variables $\mathtt{ok}$ and $\mathtt{ok}'$ are introduced into the input and output alphabets respectively to represent termination. $\mathtt{ok}$ represents the proper start of a computation, while $\mathtt{ok}'$ represents its successful termination. A design [16] is a binary relation satisfying the healthiness condition of $\mathtt{ok}$: $A = (\mathtt{ok} \Rightarrow A)$ and the downwards-closure healthiness condition of $\mathtt{ok}'$:

$$A[\mathtt{ok}' := \mathtt{false}] \Rightarrow A[\mathtt{ok}' := \mathtt{true}]\,.$$

In general, a design can be written as $b \vdash P$ where b is the weakest condition under which the computation terminates and P is a binary relation describing the constraints between the observable variables achieved on termination. Both b and P contain neither $\mathtt{ok}$ nor $\mathtt{ok}'$.

The following healthiness condition of the trace record of CSP processes defined in [16] shows how complicated a healthiness condition can be:

$$A = \prod_{s} A[tr, tr' := s,\ s \frown (tr' - tr)]$$

where $\frown$ denotes concatenation of sequences. It is not immediately clear what the condition says. In fact it states that the behaviour of a reactive process does not depend on any behaviour before the start of this process. Its corresponding predicate function does not allow straightforward proofs for its idempotence and commutativity with other healthiness functions.

The simplest form of parallelism that [16] (Chapter 7) tackles is called *disjoint parallelism*. If $\mathtt{ok}$ and $\mathtt{ok}'$ are the only free variables shared by $b \vdash P$ and $c \vdash Q$, their parallel composition becomes $b \wedge c \vdash P \wedge Q$. That means chaos (or non-termination) from either side will lead to a chaotic parallel composition; however if both sides terminate successfully, the parallel composition also terminates and achieves both P and Q in conjunction.

Based on disjoint parallelism, a more general parallel composition called a *parallel-by-merge* is defined [16]. In a parallel-by-merge composition, two components A and B are first simulated separately under disjoint alphabets by re-labeling. Then the results from both sides of the composition are combined and exported to a merge operation following them. Suppose m is the only program variable shared by two programs A and B. Their parallel composition is defined by

$$A \parallel_M B \mathrel{\widehat{=}} (A[m'{:=}0.m'] \parallel B[m'{:=}1.m'])_{+m} ; M(\mathtt{ok}, m, 0.m, 1.m, m', \mathtt{ok}')$$

where merge M is a design, $0.m'$ and $1.m'$ are new dashed variables, and subscript $+m$ makes a copy of m: $P_{+m} \mathrel{\widehat{=}} P \wedge (m = m')$. If M is symmetric, i.e.

$$M = (0.m, 1.m := 1.m, 0.m ; M),$$

the definition of parallel composition becomes commutative. If M is associative, i.e.

$$M_3 = (0.m, 1.m, 2.m := 1.m, 2.m, 0.m) ; M_3 ,$$

where M_3 is a three-way merge relation based on M

$$M_3 \mathrel{\widehat{=}} \exists x, t \cdot M(\mathtt{ok}, m, 0.m, 1.m, x, t) \wedge M(t, m, x, 2.m, m', \mathtt{ok}'),$$

the parallel composition also becomes associative. We will be able to provide an alternative view of parallel composition in Section 3.

Defining parallelism based on sequential computation may not be general enough for a real parallel language. It is assumed that the interface connecting two components and the merge is the set of output variables shared by both components. This may turn out to be too strict. Sometimes unshared variables may need to be manipulated to meet a healthiness condition [5] ; if a shared output variable is supposed to have the same value on both sides, there will be no need to include it in the interface (e.g. the conjunctive composition in Section 5.1); and sometimes we may need to include an input variable in the interface to resolve any conflict between the inputs (e.g. the Petri-net composition in Section 5.1). Setting the interface freely and explicitly will immediately solve these problems. Details can be found in Section 3 and 5.

3 Generic composition

As motivation we observe that various compositional definitions and designs tend to have a similar pattern. For example,

- in relational semantics [21], the sequential composition of two binary relations r and s is defined by $x\,(r;s)\,z \mathrel{\widehat{=}} \exists y \cdot x\,r\,y \wedge y\,s\,z$;
- in CSP ([13] Section 4.3), the parallel composition in most applications is defined by $((c!v \to P) \| (c?x \to Q(x))) \setminus C = (P \| Q(v)) \setminus C$ where communications along channel c are concealed[1];
- in CCS ([17] Section 2.2), two components A and B with ports c and $\overline{c}$ respectively can be combined and they communicate through the ports according to a rule $\frac{A \xrightarrow{c} A',\ B \xrightarrow{\overline{c}} B'}{A|B \xrightarrow{\tau} A'|B'}$ in which each communication becomes a silent action not directly observable from the outside.

In general, a system combining two components A and B demonstrates the behaviours of both A *and* B, and communications between the two components are normally delivered through an interface hidden from the outside. Generic composition provides a general form of such compositionality based on first-order logic.

3.1 Definition

A basic generic composition is written as $A :_\beta B$ in which β, called the *interface*, is a set of variables without overline decorations, and A and B can be any predicates with any free variables. A generic composition hides the interface between its components by applying an existential quantifier to the interface.

Definition 1 $\quad A :_\beta B \mathrel{\widehat{=}} \exists \boldsymbol{\beta}_0 \cdot A[\boldsymbol{\beta} := \boldsymbol{\beta}_0] \wedge B[\overline{\boldsymbol{\beta}} := \boldsymbol{\beta}_0]$

In this definition, we suppose $\boldsymbol{\beta}_0$ is a list of *new* variables free in neither A nor B and the interface β does not contain any overlined variable. The definition looks superficially like sequential composition. However it does not distinguish input and output variables; its interface is set explicitly; and there is no alphabetical restriction.

In this paper, we shall use $A = \exists x \cdot A$ to denote the condition that predicate A *does not depend on* variable x. For example, neither predicate $(y > 2)$ nor $(x = x \wedge y > 2)$ depend on x. In the latter example x is actually a free variable, although the predicate does not state anything interesting about x. Such variable independence is more general than syntactical restriction imposed by an alphabet ([16] Chapter 1). Interestingly $A = \forall x \cdot A$ states the same healthiness condition.

Thus the alphabetical restriction can now be replaced by a series of healthiness conditions:

$$A = \exists x \cdot A \qquad\qquad (x \notin \alpha).$$

All these healthiness functions are idempotent and commutative, therefore they can be considered as a *grand healthiness condition*. Whenever we need an alphabet for a predicate, we can simply impose this healthiness condition on it.

[1] Two CSP processes $a \to A$ and $a \to B$ may also communicate with each other through the shared event a.

The following propositions indicate some simple cases of basic generic composition and are proved by routine predicate manipulations.

Proposition 1 $A \wedge B = A :_\emptyset B$

Proposition 2 $\exists \boldsymbol{\beta} \cdot A = A :_\beta true$

Proposition 3 $A[\boldsymbol{\beta} := \boldsymbol{\gamma}] = A :_\beta (\overline{\boldsymbol{\beta}} = \boldsymbol{\gamma})$

3.2 Weak inverse

Most arithmetic operations have an inverse operation. For example $-$ is the inverse of $+$ in the sense that if $x + p = q$, we can restore x by calculating $q - p$.

Not every operation has an inverse: sequential composition does not. In general X cannot be restored from $X\,;\, P = Q$. Nevertheless it is possible to define sequential composition's *weak* inverse Q/P, which is the weakest specification refining Q if followed by P

$$Q/P \mathrel{\widehat{=}} \sqcap_Y \ (Y\,;\, P \sqsupseteq Q).$$

This concept (Hoare and He [14]) is called the *weakest prespecification*. A weak inverse of parallel composition called *weakest environment* was defined by Lai and Sanders [19]. It was applied to Valiant's BSP [23] by Lecomber [20].

A similar weak inverse operation can be defined for generic composition. We call it *the weakest pre-predicate*. Given two predicates Q and P, we define $Q/_\beta P$ to be the weakest predicate making $X :_\beta P \Rightarrow Q$ always true.

The following Galois connection provides a neat definition as a weak inverse in the sense of adjunctions.

Definition 2 $Q \sqsubseteq X :_\beta P$ *iff* $Q/_\beta P \sqsubseteq X$.

Theorem 1 $Q/_\beta P = \neg(\neg Q :_\beta P[\boldsymbol{\beta}, \overline{\boldsymbol{\beta}} := \overline{\boldsymbol{\beta}}, \boldsymbol{\beta}])$

Now we can express some more primitive combinators; again the proofs are routine predicate manipulations.

Proposition 4 $A \Rightarrow B = B/_\emptyset A$

Proposition 5 $\neg A = false/_\emptyset A$

Proposition 6 $A \vee B = B/_\emptyset (false/_\emptyset A)$

Proposition 7 $\forall \boldsymbol{\beta} \cdot A = A/_\beta true$

Theorem 2 (Completeness) *Any predicate that does not contain overlined free variables can be written in terms of generic composition and its weak inverse using only the constant predicates* `true` *and* `false` *and predicate letters.*

This is justified by the fact that by Propositions 1 to 7 all logic operators of predicate calculus can be eliminated from a predicate by the propositions of generic composition and its weak inverse.

The completeness theorem shows the potential of generic composition in expressing and defining semantic operators, which we consider in section 4.

3.3 Laws

We shall use a convention $x \diamond y$ for disjointness sets: $x \diamond y \mathrel{\widehat{=}} (x \cap y = \emptyset)$. Only non-trivial proofs are given.

Law 1 (Associativity) $\quad (A :_\beta B) :_\beta C = A :_\beta (B :_\beta C)$

The following law reveals an interesting precondition for general associativity. law 1 is a special case of law 2 when $\alpha = \gamma = \emptyset$.

Law 2 (Associativity′) *Assume that* $\alpha \diamond \beta$, $\beta \diamond \gamma$, $\gamma \diamond \alpha$, $A = \exists \boldsymbol{\gamma} \cdot A$ *and* $C = \exists \overline{\boldsymbol{\alpha}} \cdot C$. *Then* $(A :_{\alpha \cup \beta} B) :_{\beta \cup \gamma} C = A :_{\alpha \cup \beta} (B :_{\beta \cup \gamma} C)$.

Proof.

$$
\begin{aligned}
& (A :_{\alpha \cup \beta} B) :_{\beta \cup \gamma} C \\
= \;& \exists \boldsymbol{\beta}_1 \boldsymbol{\gamma}_1 \cdot (\exists \boldsymbol{\alpha}_0 \boldsymbol{\beta}_0 \cdot (A[\boldsymbol{\alpha}_0/\boldsymbol{\alpha}, \boldsymbol{\beta}_0/\boldsymbol{\beta}] \wedge B[\boldsymbol{\alpha}_0/\overline{\boldsymbol{\alpha}}, \boldsymbol{\beta}_0/\overline{\boldsymbol{\beta}}])[\boldsymbol{\beta}_1/\boldsymbol{\beta}, \boldsymbol{\gamma}_1/\boldsymbol{\gamma}] \\
& \quad \wedge C[\boldsymbol{\beta}_1/\overline{\boldsymbol{\beta}}, \boldsymbol{\gamma}_1/\overline{\boldsymbol{\gamma}}]) \\
= \;& \exists \boldsymbol{\beta}_1 \boldsymbol{\gamma}_1 \boldsymbol{\alpha}_0 \boldsymbol{\beta}_0 \cdot (A[\boldsymbol{\alpha}_0/\boldsymbol{\alpha}, \boldsymbol{\beta}_0/\boldsymbol{\beta}] \wedge B[\boldsymbol{\beta}_1/\boldsymbol{\beta}, \boldsymbol{\gamma}_1/\boldsymbol{\gamma}][\boldsymbol{\alpha}_0/\overline{\boldsymbol{\alpha}}, \boldsymbol{\beta}_0/\overline{\boldsymbol{\beta}}] \\
& \quad \wedge C[\boldsymbol{\beta}_1/\overline{\boldsymbol{\beta}}, \boldsymbol{\gamma}_1/\overline{\boldsymbol{\gamma}}][\boldsymbol{\alpha}_0/\overline{\boldsymbol{\alpha}}, \boldsymbol{\beta}_0/\overline{\boldsymbol{\beta}}]) \\
= \;& \exists \boldsymbol{\alpha}_0 \boldsymbol{\beta}_0 \cdot (A[\boldsymbol{\alpha}_0/\boldsymbol{\alpha}, \boldsymbol{\beta}_0/\boldsymbol{\beta}] \wedge (B :_{\beta \cup \gamma} C)[\boldsymbol{\alpha}_0/\overline{\boldsymbol{\alpha}}, \boldsymbol{\beta}_0/\overline{\boldsymbol{\beta}}]) \\
= \;& A :_{\alpha \cup \beta} (B :_{\beta \cup \gamma} C)
\end{aligned}
$$

□

Law 3 (Distributivity of disjunction)

$$
\begin{aligned}
(A \vee B) :_\beta C &= (A :_\beta C) \vee (B :_\beta C) \\
A :_\beta (B \vee C) &= (A :_\beta B) \vee (A :_\beta C)
\end{aligned}
$$

$\overline{\boldsymbol{\beta}} = \boldsymbol{\beta}$ is called the *β-identity predicate*. It forms a left and right zero of generic composition.

Law 4 (New interface)

$$
\begin{aligned}
A &= A :_\beta (\overline{\boldsymbol{\beta}} = \boldsymbol{\beta}) \\
A &= (\overline{\boldsymbol{\beta}} = \boldsymbol{\beta}) :_\beta A
\end{aligned}
$$

Law 5 (Conjunction elimination)

$$
\begin{aligned}
A \wedge (A :_\beta B) &= A :_\beta (\overline{\boldsymbol{\beta}} = \boldsymbol{\beta} \wedge B) \\
(A :_\beta B) \wedge B &= (\overline{\boldsymbol{\beta}} = \boldsymbol{\beta} \wedge A) :_\beta B
\end{aligned}
$$

Law 6 (Disjunction elimination)

$$A \vee (A :_\beta B) = A :_\beta (\overline{\boldsymbol{\beta}} = \boldsymbol{\beta} \vee B)$$
$$(A :_\beta B) \vee B = (\overline{\boldsymbol{\beta}} = \boldsymbol{\beta} \vee A) :_\beta B$$

Law 7 (Conjunction merge) *Assume* $C = \exists \boldsymbol{\beta} \cdot C$ *and* $D = \exists \overline{\boldsymbol{\beta}} \cdot D$. *Then*

$$(A :_\beta B) \wedge C = A :_\beta (B \wedge C)$$
$$D \wedge (A :_\beta B) = (D \wedge A) :_\beta B .$$

The interface of a generic composition can be replaced by some new variables.

Law 8 (Interface re-labeling) *Assume* $A = \exists \boldsymbol{\gamma} \exists \overline{\boldsymbol{\gamma}} \cdot A$ *and* $B = \exists \boldsymbol{\gamma} \exists \overline{\boldsymbol{\gamma}} \cdot B$. *Then*

$$A :_\beta B = A[\boldsymbol{\beta} := \boldsymbol{\gamma}] :_\gamma B[\overline{\boldsymbol{\beta}} := \overline{\boldsymbol{\gamma}}] .$$

Law 9 (Interface expansion) *Assume* $A = \exists \boldsymbol{\gamma} \cdot A$ *and* $B = \exists \overline{\boldsymbol{\gamma}} \cdot B$. *Then*

$$A :_\beta B = A :_{\beta \setminus \gamma} B = A :_{\beta \cup \gamma} B .$$

Law 10 (Interface extension) *Assume* $\gamma \diamond \beta$. *Then*

$$A :_\beta \exists \overline{\boldsymbol{\gamma}} \cdot B = A :_{\beta \cup \gamma} (\overline{\boldsymbol{\gamma}} = \boldsymbol{\gamma} \wedge \exists \overline{\boldsymbol{\gamma}} \cdot B)$$
$$\exists \boldsymbol{\gamma} \cdot A :_\beta B = (\overline{\boldsymbol{\gamma}} = \boldsymbol{\gamma} \wedge \exists \boldsymbol{\gamma} \cdot A) :_{\beta \cup \gamma} B .$$

Proof.
$$\begin{aligned} & A :_{\beta \cup \gamma} (\overline{\boldsymbol{\gamma}} = \boldsymbol{\gamma} \wedge \exists \overline{\boldsymbol{\gamma}} \cdot B) \\ = {} & \exists \boldsymbol{\beta}_0 \boldsymbol{\gamma}_0 \cdot A[\boldsymbol{\beta}_0/\boldsymbol{\beta}, \boldsymbol{\gamma}_0/\boldsymbol{\gamma}] \wedge (\boldsymbol{\gamma}_0 = \boldsymbol{\gamma}) \wedge \exists \overline{\boldsymbol{\gamma}} B[\overline{\boldsymbol{\beta}} := \boldsymbol{\beta}_0] \\ = {} & \exists \boldsymbol{\beta}_0 \cdot A[\boldsymbol{\beta} := \boldsymbol{\beta}_0] \wedge \exists \overline{\boldsymbol{\gamma}} B[\overline{\boldsymbol{\beta}} := \boldsymbol{\beta}_0] \\ = {} & A :_\beta \exists \overline{\boldsymbol{\gamma}} \cdot B. \end{aligned}$$
The second part of this law can be proved similarly. □

The following law describes a sufficient condition under which two interfaces β and γ will not interfere with each other, and therefore components B and C are actually placed in conjunction and can be connected to an existing component A in either order. Thus components B and C can be considered to be in parallel. Parallel-via-medium in Section 4 is defined by this law.

Law 11 (Left/right interface commutativity) *Assume* $\gamma \diamond \beta$, $B = \exists \boldsymbol{\gamma} \exists \overline{\boldsymbol{\gamma}} \cdot B$ *and* $C = \exists \boldsymbol{\beta} \exists \overline{\boldsymbol{\beta}} \cdot C$. *Then*

$$(A :_\beta B) :_\gamma C = A :_{\beta \cup \gamma} (B \wedge C) = (A :_\gamma C) :_\beta B$$
$$C :_\gamma (B :_\beta A) = (C \wedge B) :_{\beta \cup \gamma} A = B :_\beta (C :_\gamma A) .$$

Proof.
$$\begin{aligned} & (A :_\beta B) :_\gamma C \\ = {} & A :_{\beta \cup \gamma} (\overline{\boldsymbol{\gamma}} = \boldsymbol{\gamma} \wedge B) :_{\beta \cup \gamma} (\overline{\boldsymbol{\beta}} = \boldsymbol{\beta} \wedge C) \\ = {} & A :_{\beta \cup \gamma} ((\overline{\boldsymbol{\gamma}} = \boldsymbol{\gamma} \wedge B) :_\gamma C) \\ = {} & A :_{\beta \cup \gamma} (B \wedge C) \\ = {} & (A :_\gamma C) :_\beta B. \end{aligned}$$
The second part of this law can be proved similarly. □

law 12 is a special case of Law 11 when $\gamma = \emptyset$. If component C has no interference with the interface of another generic composition in conjunction, it can be located anywhere in the composition.

Law 12 (Commutativity with conjunction) *Assume* $C = \exists \boldsymbol{\beta} \exists \overline{\boldsymbol{\beta}} \cdot C$. *Then*

$$(A :_\beta B) \wedge C = (A \wedge C) :_\beta B = A :_\beta (B \wedge C).$$

If the interfaces of two generic compositions in a conjunction have no interference with each other, they can be merged into a joint interface.

Law 13 (Interface merge) *Assume* $A = \exists \boldsymbol{\gamma} \cdot A$, $B = \exists \overline{\boldsymbol{\gamma}} \cdot B$, $C = \exists \boldsymbol{\beta} \cdot C$ *and* $D = \exists \overline{\boldsymbol{\beta}} \cdot D$. *Then* $(A :_\beta B) \wedge (C :_\gamma D) = (A \wedge C) :_{\beta \cup \gamma} (B \wedge D)$.

Proof.

1. If $\beta \diamond \gamma$, then
   $(A :_\beta B) \wedge (C :_\gamma D)$
   $= (\exists \boldsymbol{\beta}_0 \cdot A[\boldsymbol{\beta} := \boldsymbol{\beta}_0] \wedge B[\overline{\boldsymbol{\beta}} := \boldsymbol{\beta}_0]) \wedge (\exists \boldsymbol{\gamma}_0 \cdot C[\boldsymbol{\gamma} := \boldsymbol{\gamma}_0] \wedge D[\overline{\boldsymbol{\gamma}} := \boldsymbol{\gamma}_0])$
   $= \exists \boldsymbol{\beta}_0 \exists \boldsymbol{\gamma}_0 \cdot A[\boldsymbol{\beta} := \boldsymbol{\beta}_0] \wedge B[\overline{\boldsymbol{\beta}} := \boldsymbol{\beta}_0] \wedge C[\boldsymbol{\gamma} := \boldsymbol{\gamma}_0] \wedge D[\overline{\boldsymbol{\gamma}} := \boldsymbol{\gamma}_0]$
   $= (A \wedge C) :_{\beta \cup \gamma} (B \wedge D)$.
2. If $\beta \cap \gamma = \phi \neq \emptyset$ and $\varphi \mathrel{\widehat{=}} \beta \setminus \phi$, then $A :_\beta B = A :_\varphi B$ according to law 9. □

4 Defining other semantic operators

4.1 Sequential composition

A sequential computation is a predicate with input alphabet γ and output alphabet β'. The input alphabet contains all observable program variables and the output alphabet contains all modifiable program variables. We suppose any modifiable program variable has its corresponding observable variable, i.e. $\beta \subseteq \gamma$. Then sequential composition can be defined by generic composition.

Definition 3 $A ; B \mathrel{\widehat{=}} A :_{\beta'} B[\boldsymbol{\beta} := \overline{\boldsymbol{\beta}}']$

Interestingly, it is also possible to define it in the reverse order. Note that, in a generic composition $X :_\beta Y$, any variable $x \in \beta$ in X corresponds to $\overline{x}$ in Y.

Theorem 3 $A ; B = B :_\beta A[\boldsymbol{\beta}' := \overline{\boldsymbol{\beta}}]$

Thus sequential composition is a special kind of generic composition where the interface is set to consist of all output variables. Its properties can be easily deduced from the laws in section 3.3.

4.2 Parallel composition via medium

Parallel-via-medium composition is defined as follows:

Definition 4 $A \overset{M}{\|}_\pi B \mathrel{\widehat{=}} (A[\boldsymbol{\pi} := \boldsymbol{\pi}.0] \wedge B[\boldsymbol{\pi} := \boldsymbol{\pi}.1]) :_{\pi.0 \cup \pi.1} M$

Variable set π explicitly sets the interface between A and B. If components A and B do not depend on $\pi.0$ and $\pi.1$, $\boldsymbol{\pi}$ will be substituted by two lists of *new* variables $\boldsymbol{\pi}.0$ and $\boldsymbol{\pi}.1$ in A and B, respectively. In case A or B depends on $\pi.0$ or $\pi.1$, law 8 can be used to re-label the interface with some new variables to avoid interference.

M, called a *medium* between A and B, is a predicate with any free variables. A medium puts all information from $\pi.0$ and $\pi.1$ together and restores interface π. To make a parallel-via-medium composition commutative and associative, a medium needs to satisfy corresponding healthiness conditions.

The convention $\Pi_i \mathrel{\widehat{=}} (\overline{\boldsymbol{\pi}}.i = \boldsymbol{\pi})$ is used in the following theorems. Thus we have $M = \Pi_0 \overset{M}{\|}_\pi \Pi_1$. We also assume that A, B and C do not depend on any variable in $\pi.i$ or $\overline{\pi}.i$ $(i = 0, 1, \cdots)$.

Theorem 4 *For any A and B, $A \overset{M}{\|}_\pi B = B \overset{M}{\|}_\pi A$ iff M is commutative in this sense: $\Pi_0 \overset{M}{\|}_\pi \Pi_1 = \Pi_1 \overset{M}{\|}_\pi \Pi_0$.*

Theorem 5 *For any A, B and C, $(A \overset{M}{\|}_\pi B) \overset{M}{\|}_\pi C = A \overset{M}{\|}_\pi (B \overset{M}{\|}_\pi C)$ iff M is associative in this sense: $(\Pi_0 \overset{M}{\|}_\pi \Pi_1) \overset{M}{\|}_\pi \Pi_2 = \Pi_0 \overset{M}{\|}_\pi (\Pi_1 \overset{M}{\|}_\pi \Pi_2)$.*

Proof. Let $A_0 \mathrel{\widehat{=}} [\boldsymbol{\pi} := \boldsymbol{\pi}.0]$, $B_0 \mathrel{\widehat{=}} [\boldsymbol{\pi} := \boldsymbol{\pi}.1]$ and $C_0 \mathrel{\widehat{=}} [\boldsymbol{\pi} := \boldsymbol{\pi}.2]$. If M is associative then

$$
\begin{aligned}
(A \overset{M}{\|}_\pi B) \overset{M}{\|}_\pi C &= (A_0 \wedge B_1 :_{\pi.0 \cup \pi.1} \Pi_0 \overset{M}{\|}_\pi \Pi_1) \overset{M}{\|}_\pi C \\
&= (A_0 \wedge B_1) :_{\pi.0 \cup \pi.1} (\Pi_0 \overset{M}{\|}_\pi \Pi_1) \overset{M}{\|}_\pi (C_2 :_{\pi.2} \Pi_2) \\
&= (A_0 \wedge B_1) :_{\pi.0 \cup \pi.1} (C_2 :_{\pi.2} (\Pi_0 \overset{M}{\|}_\pi \Pi_1) \overset{M}{\|}_\pi \Pi_2) \\
&= (A_0 \wedge B_1 \wedge C_2) :_{\pi.0 \cup \pi.1 \cup \pi.2} (\Pi_0 \overset{M}{\|}_\pi \Pi_1) \overset{M}{\|}_\pi \Pi_2 \\
&= (A_0 \wedge B_1 \wedge C_2) :_{\pi.0 \cup \pi.1 \cup \pi.2} \Pi_0 \overset{M}{\|}_\pi (\Pi_1 \overset{M}{\|}_\pi \Pi_2) \\
&= (B_1 \wedge C_2) :_{\pi.1 \cup \pi.2} (A_0 :_{\pi.0} \Pi_0 \overset{M}{\|}_\pi (\Pi_1 \overset{M}{\|}_\pi \Pi_2)) \\
&= (B_1 \wedge C_2) :_{\pi.1 \cup \pi.2} (A_0 :_{\pi.0} \Pi_0) \overset{M}{\|}_\pi (\Pi_1 \overset{M}{\|}_\pi \Pi_2) \\
&= A \overset{M}{\|}_\pi (B_1 \wedge C_2 :_{\pi.1 \cup \pi.2} \Pi_1 \overset{M}{\|}_\pi \Pi_2) \\
&= A \overset{M}{\|}_\pi (B \overset{M}{\|}_\pi C).
\end{aligned}
$$

The other direction of the theorem is proved by assigning A, B and C to be Π_0, Π_1 and Π_2 respectively. □

Parallel-via-medium has one more parameter π. It may include either shared or unshared output variables, input variables or their mixture. However, if π is the set of all shared output variables of both components, the composition becomes a parallel-by-merge composition [16].

4.3 Higher-order healthiness conditions

A higher-order healthiness condition defined by generic composition is normally written in the form $A = A :_\beta R$ and called a *generic healthiness condition*. Its corresponding healthiness function is $\lambda X \cdot X :_\beta R$.

Writing a healthiness condition in this form is motivated by [16] (Chapter 3) in which the condition **H3** for the total-correctness of designs is defined by $A = A ; \Pi$ where Π denotes the 'skip' command. Besides the simplicity of this condition, the proof of its idempotence is pleasingly straightforward $\Pi ; \Pi = \Pi$. As we discussed in the introduction, the proofs of idempotence and commutativity can be otherwise extremely difficult for a real language [5]. This technique has provided a simple solution and a uniform way of writing and reasoning about higher-order healthiness conditions.

Most healthiness conditions cannot be defined in this form with a sequential composition. Fortunately, using generic composition, we can generalize this technique and tackle a variety of more complicated healthiness conditions (see Section 5).

In a real semantics, there can be a mixture of conjunctive, disjunctive and generic healthiness conditions. The following theorems study the conditions under which a healthiness function based on generic composition is idempotent and commute with other healthiness functions.

Theorem 6 *$\lambda X \cdot X :_\beta R$ is an idempotent function iff $R :_\beta R = R$.*

Theorem 7 *If $R = \exists \boldsymbol{\gamma} \exists \overline{\boldsymbol{\gamma}} \cdot R$ and $S = \exists \boldsymbol{\beta} \exists \overline{\boldsymbol{\beta}} \cdot S$, then $\lambda X \cdot X :_\beta R$ and $\lambda X \cdot X :_\gamma S$ commute.*

Theorem 8 *If $S = \exists \boldsymbol{\beta} \exists \overline{\boldsymbol{\beta}}\, S$, then $\lambda X \cdot X :_\beta R$ and $\lambda X \cdot X \wedge S$ commute.*

Theorem 9 *If $S = S :_\beta R$, then $\lambda X \cdot X :_\beta R$ and $\lambda X \cdot X \vee S$ commute.*

Proof.

$$\begin{aligned}
&(\lambda X \cdot X :_\beta R) \circ (\lambda X \cdot X \vee S) \\
&= \lambda X \cdot (X \vee S) :_\beta R \\
&= \lambda X \cdot (X :_\beta R) \vee (S :_\beta R) \\
&= \lambda X \cdot (X :_\beta R) \vee S \\
&= (\lambda X \cdot X \vee S) \circ (\lambda X \cdot X :_\beta R).
\end{aligned}$$

□

Theorem 10 *If $S = S :_\beta R$, then $\lambda X \cdot X :_\beta R$ and $\lambda X \cdot X \lhd \exists \boldsymbol{\beta}\, b \rhd S$ commute.*

In this definition, the binary conditional $A \lhd b \rhd B$ is defined as $(b \wedge A) \vee (\neg b \wedge B)$.

4.4 Two-dimensional generic composition

A parallel-via-medium composition in which the components on both sides are identical becomes a more general kind of composition called a *two-dimensional generic composition.*

Definition 5 *The construct* $A ::_\beta B \mathrel{\widehat{=}} A \overset{B}{\|}_\beta A$ *is called a two-dimensional generic composition.*

If we consider A as a component, a two-dimensional generic composition with another component B compares any two behaviours of A by allowing only some of the observable variables (i.e. in β) to be different.

This composition enjoys some laws (e.g. associativity) similar to those of basic generic composition in Section 3.3 and Section 4.3.

A healthiness condition based on this composition is normally written in the form $A = A ::_\beta R$. The significance is that the component A in the definition appears twice; thus it makes it possible to compare any two values of the variables in $\boldsymbol{\beta}$.

This technique is highly useful when we need to impose a healthiness condition on the *range* of some variable. A range is the set of the non-deterministic choices of a variable when other variables are fixed. For example, the range of y in predicate $(y \geqslant x + 1)$ is $\{n \in \mathbb{N} \mid n \geqslant 5\}$ if x is fixed at 4.

4.5 Multi-dimensional generic composition

We take one further step to allow up-to-infinitely-many copies of a component to appear in a predicate. Note that the following definition may contain an infinite number of variables.

Definition 6 $A :_{\beta^I} B \mathrel{\widehat{=}} \left(\bigwedge_{i \in I} A[\boldsymbol{\beta} := \boldsymbol{\beta}.i]\right) :_{\bigcup_{i \in I} \beta.i} B$

In the definition, I is called an *indexing set.* Each index i in I corresponds to a copy of A whose variables in $\boldsymbol{\beta}$ are substituted by those in $\boldsymbol{\beta}.i$. This composition enjoys some laws similar to those in Section 3.3 and Section 4.3.

If $I = \{0, 1\}$, this definition collapses to a two-dimensional generic composition. However if I is an infinite set, interface $\{\boldsymbol{\beta}_i \mid i \in I\}$ from the copies of A will form a subset (cardinality up to that of I) of all the possible non-deterministic choices; if I is the Cartesian product of the types of the variables from β, the interface can form any subset of the non-deterministic choices.

A healthiness condition based on this composition is normally written in the form $A = A :_{\beta^I} R$. The significance of this composition is that now we can impose healthiness restrictions on the cardinality or limit points of a variable's range.

5 Applications

In this section we shall demonstrate the expressive power of generic composition by applying it to yield a variety of compositions and higher-order healthiness conditions.

5.1 Parallel compositions

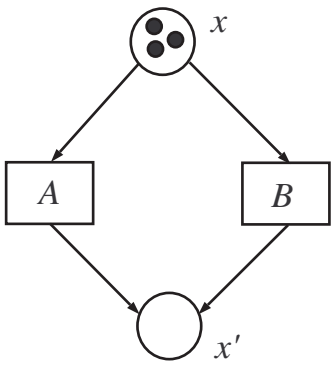
5.1.1 [Conjunction]

A (binary) parallel composition demonstrates the common behaviour of two components and hence becomes a conjunction if the interface between the two components is empty.

Proposition 8 $A \wedge B = A \overset{\mathtt{true}}{\|}_{\emptyset} B$

5.1.2 [Interference between the inputs]

In a parallel-by-merge composition, the merge is composed sequentially after the simulation of the two components. That means the initial states of the two components are simply the direct copies of the initial state of the parallel composition. Consequently no interference is allowed at the beginning of a parallel composition. Such a definition may not be general enough in some cases. For example, in the following Petri net, x and x' denote the initial and final states, respectively. Place x contains a number of tokens. A token in x can be received by either process A or process B, which are represented in the abstract by two transitions.



It is not compatible with parallel-by-merge in that the number of tokens received by process A (denoted by $x.\mathtt{0}$) does not only depend on the number of tokens in x; instead it also depends on the number of tokens received by process B (denoted by $x.\mathtt{1}$). Similarly the tokens output to x' include those tokens from either A (denoted by $x'.\mathtt{0}$) or B (denoted by $x'.\mathtt{1}$).

Fortunately we can define it as a parallel-via-medium composition with an interface containing both input and output variables. Thus the medium is actually located between the two components.

Definition 7 $A \| B \mathrel{\widehat{=}} A \overset{M}{\|}_{\{x,x'\}} B$ *where*

$$M \mathrel{\widehat{=}} (x = \overline{x}.\mathtt{0} + \overline{x}.\mathtt{1}) \wedge (x' = \overline{x}'.\mathtt{0} + \overline{x}'.\mathtt{1}).$$

5.1.3 [Interleaving composition]

Another interesting parallel composition is the fair interleaving of non-terminating reactive processes. The predicative semantics of a reactive process is an infinite sequence (called *a trace*) consisting of observable intermediate events of the process. We let t be a variable representing a trace, whose type is $\mathbb{N} \to \mathbb{T}$ where $\mathbb{T}$ is the type of each event.

The semantics of $A \,|||\, B$ is a trace consisting of *all* intermediate events of A and B in any interleaved order. Fairness is guaranteed since every event will eventually appear in the composition's trace.

Definition 8 $A \,|||\, B \mathrel{\widehat{=}} A \overset{M}{\parallel}_{\{t\}} B$ *where*

$$M \mathrel{\widehat{=}} \exists f, g : \mathtt{mon}(\mathbb{N} \to \mathbb{N}) \cdot \text{fill}\,\{f, g\} \wedge (t \circ f \equiv \overline{t}.\mathtt{0}) \wedge (t \circ g \equiv \overline{t}.\mathtt{1}).$$

In the above definition, $\mathtt{mon}(\mathbb{N} \to \mathbb{N})$ is the set of all strictly monotonic functions on natural numbers: $i \prec j \Rightarrow f(i) \prec f(j)$. $\overline{t}.\mathtt{0}$, $\overline{t}.\mathtt{1}$ and t are the traces of A, B and $A \,|||\, B$, respectively. 'fill' is a predicate on a set of functions whose ranges form a partition of all natural numbers:

$$\begin{aligned} \text{fill}\,\mathcal{F} \mathrel{\widehat{=}}\ & (\mathbb{N} = \bigcup\{\mathtt{ran}\ f \mid f \in \mathcal{F}\}) \\ & \wedge\ \forall f, g : \mathcal{F} \cdot (\mathtt{ran}\ f \diamond \mathtt{ran}\ g) \vee (\mathtt{ran}\ f = \mathtt{ran}\ g) \end{aligned}$$

Functions f and g map every variable in $\overline{t}.\mathtt{0}$ and $\overline{t}.\mathtt{1}$ to a variable in t. The original order of the variables is preserved by the monotonicity of f and g. Thus t can be any fair interleaving of $\overline{t}.\mathtt{0}$ and $\overline{t}.\mathtt{1}$.

The commutativity of this composition is proved by Theorem 4 and the symmetry of the medium in Definition 8. Its associativity is proved as follows:

Proof.

$$(\Pi_0 \overset{M}{\parallel}_{\{t\}} \Pi_1) \overset{M}{\parallel}_{\{t\}} \Pi_2$$

$$= \begin{pmatrix} \exists f_1, g_1 : \mathtt{mon}(\mathbb{N} \to \mathbb{N}) \cdot \\ \quad \text{fill}\,\{f_1, g_1\} \\ \quad \wedge\, t.\mathtt{0} \circ f_1 \equiv \overline{t}.\mathtt{0} \\ \quad \wedge\, t.\mathtt{0} \circ g_1 \equiv \overline{t}.\mathtt{1} \\ \quad \wedge\, t.\mathtt{1} \equiv \overline{t}.\mathtt{2} \end{pmatrix} :_{\{t.\mathtt{0}, t.\mathtt{1}\}} \begin{pmatrix} \exists f_2, g_2 : \mathtt{mon}(\mathbb{N} \to \mathbb{N}) \cdot \\ \quad \text{fill}\,\{f_2, g_2\} \\ \quad \wedge\, t \circ f_2 \equiv \overline{t}.\mathtt{0} \\ \quad \wedge\, t \circ g_2 \equiv \overline{t}.\mathtt{1} \end{pmatrix}$$

$$\begin{aligned} = \quad & \exists f_1, g_1, f_2, g_2 : \mathtt{mon}(\mathbb{N} \to \mathbb{N}) \cdot \text{fill}\,\{f_2 \circ f_1,\ f_2 \circ g_1,\ g_2\} \\ & \wedge (t \circ f_2 \circ f_1 \equiv \overline{t}.\mathtt{0}) \wedge (t \circ f_2 \circ g_1 \equiv \overline{t}.\mathtt{1}) \wedge (t \circ g_2 \equiv \overline{t}.\mathtt{2}) \end{aligned}$$

$$
\begin{array}{ll}
= & \exists f, g, h : \mathtt{mon}(\mathbb{N} \to \mathbb{N}) \cdot \text{ fill } \{f, g, h\} \\
& \quad \wedge (t \circ f \equiv \overline{t}.0) \wedge (t \circ g \equiv \overline{t}.1) \wedge (t \circ h \equiv \overline{t}.2)
\end{array}
$$

$$
\begin{array}{ll}
= & \exists f_1, g_1, f_2, g_2 : \mathtt{mon}(\mathbb{N} \to \mathbb{N}) \cdot \text{ fill } \{f_2,\ g_2 \circ f_1,\ g_2 \circ g_1\} \\
& \quad \wedge (t \circ f_2 \equiv \overline{t}.0) \wedge (t \circ g_2 \circ f_1 \equiv \overline{t}.1) \wedge (t \circ g_2 \circ g_1 \equiv \overline{t}.2)
\end{array}
$$

$$
= \begin{pmatrix} \exists f_1, g_1 : \mathtt{mon}(\mathbb{N} \to \mathbb{N}) \cdot \\ \quad \text{fill } \{f_1, g_1\} \\ \quad \wedge\, t.0 \equiv \overline{t}.0 \\ \quad \wedge\, t.1 \circ f_1 \equiv \overline{t}.1 \\ \quad \wedge\, t.1 \circ g_1 \equiv \overline{t}.2 \end{pmatrix} :_{\{t.0, t.1\}} \begin{pmatrix} \exists f_2, g_2 : \mathtt{mon}(\mathbb{N} \to \mathbb{N}) \cdot \\ \quad \text{fill } \{f_2, g_2\} \\ \quad \wedge\, t \circ f_2 \equiv \overline{t}.0 \\ \quad \wedge\, t \circ g_2 \equiv \overline{t}.1 \end{pmatrix}
$$

$$
= \mathtt{II}_0 \overset{M}{\parallel}_{\{t\}} (\mathtt{II}_1 \overset{M}{\parallel}_{\{t\}} \mathtt{II}_2)
$$

Thus $|||$ is associative according to theorem 5. □

The predicative semantics of a non-terminating reactive process does not have any output variables. Without the differentiation between input and output variables, no parallel-by-merge composition can be properly defined. Parallel-via-medium solves this problem by directly setting the interface to be $\{t\}$.

5.2 Healthiness conditions based on basic generic composition

In this section, we study some healthiness conditions appearing in various semantic models. Each healthiness condition is written in the form $A = A :_\beta R$, $A = A ::_\beta R$ or $A = A :_{\beta^I} R$. The corresponding monotonic and idempotent healthiness function transforms any unhealthy predicate to a healthy one.

Recall that the range of a variable in a predicate is the set of all possible values of the variable when other variables are fixed.

5.2.1 [Downwards-closure]

In semantic studies we often need to stipulate a variable x to be downwards-closed or upwards-closed. Let the type of variable x be real numbers. The following healthiness condition requires the range of x to be $(-\infty, a)$ or $(-\infty, a]$ where a can be any real number. Note that downwards closure does not require the inclusion of the 'limit point' a. We will be able to reason about limit points using infinite-dimensional generic composition later on.

HC 1 $\quad A = A :_{\{x\}} (x \leqslant \overline{x})$

The ranges of x in the following healthy predicates can be simply compared by refinement ordering. For example $(y + 1 > x) \Rightarrow (y + 1 \geqslant x) \Rightarrow (y + 2 > x) \Rightarrow (y + 2 \geqslant x)$. According to Theorem 6, the healthiness function $\lambda X \cdot X :_{\{x\}} (x \leqslant \overline{x})$ is idempotent if $(x \leqslant \overline{x}) :_{\{x\}}$

$(x \leqslant \overline{x}) = (x \leqslant \overline{x})$, which is obvious. Healthiness 1 defines a more general downwards-closure if the type of x is a partial ordering. Upwards-closure can be expressed by replacing $\leqslant$ with $\geqslant$.

5.2.2 [Comparison of non-determinism]

A similar healthiness condition states that the value of y is always a possible non-deterministic outcome of x.

HC 2 $\quad A = A :_{\{x\}} x \in \{\overline{x},\ y\}$

5.2.3 [Prefix independence]

The healthiness condition $A = \sqcap_s A[tr, tr' := s, s \frown tr' - tr]$ for traces mentioned in section 2 now becomes as simple as follows:

HC 3 $\quad A = A :_{\{tr,tr'\}} (\overline{tr}' - \overline{tr} = tr' - tr)$

It is an easy exercise to check that the corresponding healthiness function is indeed monotonic and idempotent.

5.2.4 [Healthy Designs]

Another healthiness condition $A[\mathtt{ok}' := \mathtt{false}] \Rightarrow A[\mathtt{ok}' := \mathtt{true}]$ for the designs mentioned in Section 2 is redefined:

HC 4 $\quad A = A :_{\{\mathtt{ok}'\}} (\overline{\mathtt{ok}}' \Rightarrow \mathtt{ok}')$

If the range of $\mathtt{ok}'$ in a predicate A is $\{\mathtt{false}\}$, the generic composition will transform the predicate by adding 'true' to the range. Thus a predicate is healthy in this sense iff it is either deterministic on 'true' or non-deterministic between 'true' and 'false'.

5.2.5 [Prefix closure of strategy]

In game semantics [1] a *strategy* S is a non-empty set of even-length $(2 \mid len(s))$ sequences satisfying a few healthiness conditions. One of them is the prefix-closure condition:

$$sab \in S \Rightarrow s \in S.$$

The generic-composition way of writing this is:

HC 5 $\quad A = A :_{\{s\}} (2 \mid len(s) \wedge s \leqslant \overline{s})$

The corresponding healthiness function is monotonic and idempotent and it transforms any unhealthy predicate to a healthy one by adding all even-length prefixes of each sequence.

5.3 Healthiness conditions based on two-dimensional generic composition

Sometimes the component A needs to appear twice in $f(A)$. In that case, two-dimensional generic composition is needed.

5.3.1 [Decoupled variables]

For example, in distributed computing asynchronous communication exhibits latency. If there is no causal relation established by any communication between two observable variables, we call them *decoupled* variables. The following healthiness condition states that, if x and y are *decoupled* variables, their non-deterministic choices are not directly correlated. That means the range of joint non-deterministic choices of the two variables is exactly the Cartesian product of their separate ranges.

HC 6 $\quad A = A ::_{\{x,y\}} (x \in \{\overline{x}.0, \overline{x}.1\} \wedge y \in \{\overline{y}.0, \overline{y}.1\})$

Let $\{x, y, z\}$ be the set of all free variables of A. The above healthiness condition is satisfied iff $A(x, z, y)$ *can be decomposed into two predicates* $A_1(x, z)$ *and* $A_2(z, y)$ *in conjunction*, i.e. $A(x, z, y) = A_1(x, z) \wedge A_2(z, y)$ where x is not directly related to y.

5.3.2 [Determinism]

In relational semantics [21], a variable is deterministic if its range is always a singleton set. The semantic space of deterministic relations is not a complete lattice because the glb operator (i.e. relational union) is not closed in the space. However adding the bottom 'true' to such a space will create a complete lattice, which can be expressed as:

HC 7 $\quad A = A \vee A ::_{\{x\}} (\overline{x}.0 \neq \overline{x}.1)$.

The corresponding healthiness function transforms any deterministic predicate to itself and any non-deterministic predicate to the bottom. Although this definition has an equivalent form $A = A ::_{\{x\}} (x \in \{\overline{x}.0, \overline{x}.1\} \vee \overline{x}.0 \neq \overline{x}.1)$, HC 7 looks simpler.

The range of x may be an empty set, a singleton set or the set of all possible values. Although it is not exactly a healthiness condition for pure determinism, their difference lies only at some 'extreme' points. This is a law price to pay, because what we obtain is a uniform way of writing its healthiness condition and the semantic space now becomes a complete lattice. This technique is called *Completion* and has been our standard technique to deal with non-complete-lattice semantic spaces [5].

5.3.3 [Deterministic strategies]

In game semantics [1], a player and its opponent act alternatively in a play. The player may play deterministically according to the pre-history: if both sab and sac are healthy (even-length) sequences of a strategy, they must be identical, i.e. $b = c$. This healthiness condition is defined as follows:

HC 8 $A = A \vee A ::_{\{s\}} \exists tabc\,(\overline{s}.0 = tab \wedge \overline{s}.1 = tac \wedge b \neq c)$

The corresponding healthiness function transforms any deterministic strategy to itself and any non-deterministic one to the bottom predicate 'true'. The *Innocence* condition in game semantics [1] is very similar to HC 8. Other healthiness conditions in [1] including the bracketing condition can be trivially defined as conjunctive conditions.

5.3.4 [Convexity of probabilistic distributions]

In probabilistic semantics [10], the value of a variable f is a probabilistic distribution function $f : W \to [0,1]$ where W is the set of all states. The following healthiness condition states that the distributions of variable f form a convex area in the space containing all possible distribution functions.

HC 9 $A = A ::_{\{f\}} (\exists \lambda \in [0,1] \cdot (f = \lambda * \overline{f}.0 + (1-\lambda) * \overline{f}.1))$

The corresponding healthiness function transforms a predicate by adding intermediate linear combinations of every two distributions into the range.

5.4 Healthiness conditions based on infinite-dimensional generic composition

If infinitely many copies of component A are allowed to appear in a healthiness condition, we will be able to reason about the cardinality of a range and the limit points (also called 'cluster points') in a range.

5.4.1 [Closed sets and safety properties]

Let the type of variable x be $\mathbb{X}$ where $(\mathbb{X}, \mathcal{T})$ is a topological space. The following healthiness condition requires that the range of the variable is a closed set.

HC 10 $A = A :_{\{x\}^{\mathbb{X}}} (x \in \text{closure } \Sigma_x^{\mathbb{X}})$

where $\Sigma_x^{\mathbb{X}} \mathrel{\widehat{=}} \{\overline{x}.i \mid i \in \mathbb{X}\}$.

There are as many as $\texttt{card}\ \mathbb{X}$ copies of A's appearing in this definition. Thus Σ_x^{X} can be any non-empty subset of x's range. The corresponding healthiness function of this condition transforms any predicate with not closed range to a predicate with the smallest closed range containing the original range.

In the semantic models based on temporal logic (e.g. TLA [18] and UNITY [3]), a reactive process is an infinite sequence of states, each of which has a type $\mathbb{T}$ and hence the space of all processes is $\mathbb{X} \mathrel{\widehat{=}} \mathbb{X}^{\omega}$. There is a natural topology on $\mathbb{X}$ in which each basic open set is the set of all infinite sequences sharing a common prefix and each open set is the union of basic open sets. It has been proved [2] that safety properties are exactly the closed sets of the topology, (pure) liveness properties are exactly the dense sets, and any property can be expressed as the intersection of a safety property and a liveness property.

Thus the set of all predicates satisfying HC 10 on a sequence variable x forms a semantic space for all safety properties. The corresponding healthiness function transforms any property to the strongest safety property implied by it.

5.4.2 [Dense sets and liveness properties]

A dense set intersects every non-empty open set. The following healthiness condition requires the range of variable x to be a dense set:

HC 11 $\quad A = A \wedge A :_{\{x\}^{\mathbb{X}}} \forall \mathrm{O} \in \mathcal{T} \cdot (\mathrm{O} \diamond \Sigma_x^{\mathbb{X}} \Rightarrow \mathrm{O} = \emptyset)$

There are as many as $\mathtt{card}\ \mathbb{X}$ copies of A appearing in the definition. The corresponding healthiness function (monotonic and idempotent) transforms any predicate with a dense range to itself and any unhealthy predicate to the top 'false'. Again we need the help of one extra point 'false' to make the space to become a complete lattice, because the intersection of dense sets may not be dense.

Another interesting way to write a healthiness condition for liveness is:

$$A = A :_{\{x\}^{\mathbb{X}}} x \notin (\mathtt{closure}\ \Sigma_x^{\mathbb{X}} - \Sigma_x^{\mathbb{X}})$$

Its corresponding healthiness function together with that of HC 10 decomposes any property into a pair of safety and liveness properties in conjunction. Although this function is idempotent, it is not monotonic and the corresponding semantic space is not a complete lattice.

5.4.3 [Cauchy-closure of probabilistic distributions]

HC 9 requires the range of a variable to be convex. However the boundary points (i.e. limit points) may or may not be included in the range. The Cauchy-closure healthiness condition in [10] requires those boundary points to be included. To reason about limit points, we need an infinite-dimensional generic composition. Recall that the type of a variable f is a set of $\mathbb{L}$ and $\mathbb{L}$ functions $\mathbb{F} \mathrel{\widehat{=}} \mathbb{W} \to [0,1]$ (i.e. probabilistic distribution). If $\mathbb{W}$ is a finite set, $\mathbb{F}$ is isomorphic to an n-dimensional Euclidean space where n is the cardinality of $\mathbb{W}$. Cauchy-closure is then defined by:

HC 12 $\quad A = A :_{\{f\}^{\mathbb{F}}} (\forall \varepsilon > 0 \exists i \cdot |\overline{f}.i - f| < \varepsilon)$

In the definition, $|\cdot|$ is the Euclidean distance associated with the Euclidean space. The corresponding healthiness function transforms any predicate by 'pasting' those boundary points to the range of variable f.

5.4.4 [Finitary condition of relations]

Another important healthiness condition of relational semantics [21] is called the *finitary condition*, which corresponds to the $\bigvee$-continuity condition of predicate-transformer semantics [9]. A variable is finitary if its range is either a finite set or the set of all possible values.

HC 13 $A = A \vee A :_{\{x\}^{\mathbb{N}}} (\mathtt{card}\ \Sigma_x^{\mathbb{N}} = \omega)$

There are up-to-ω number of the copies of A appearing in the definition where ω is the cardinality of $\mathbb{N}$. Its corresponding healthiness function transforms a non-finitary predicate by pasting all other possible values to the range of x.

6 Conclusions and acknowledgement

Our objective is to aid semantic studies on real programming languages. The introduction of generic composition serves this purpose by unifying various semantic definitions. The definition of parallel composition and the presentation of its associativity become more general and simpler than parallel-by-merge. The technique also provides a unified form for defining healthiness conditions. The completeness theorem shows the evidence of its generality in theory. The examples from a variety of theories have shown its generality in practice.

This paper is part of the author's D.Phil thesis. The author would like thank his supervisor J. W. Sanders for the discussions and encouragement.

References

[1] A. Abramsky, Semantics of interaction: an introduction to Game Semantics, *Proc. of the 1996 CLiCS Summer School*,ed. P. Dybjer and A. Pitts, 1-31, Cambridge Press, 1997.

[2] B. Alpern and F. B. Schneider, Defining liveness, *Information Processing Letters*, **21**: 181-185, 1985.

[3] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[4] Y. Chen, *Formal Methods for Bulk-Synchronous Programming*, DPhil. thesis, Oxford University Computing Laboratory, to appear 2001.

[5] Y. Chen, How to write a healthiness condition, *2nd International Conference on Integrated Formal Methods*, Lecture Notes in Computer Science Vol 1945: 299-317, Springer-Verlag, 2000.

[6] Y. Chen, Parallel-via-medium composition, *Constructive Methods for Parallel Programming*, ed. S. Gorlatch and C. Lengauer, in book series Advances in Computation: Theory and Practice, Nova Science Books and Journals, 2000.

[7] Y. Chen and J. W. Sanders, The weakest specifunction, *PRG Technical Report* TR-8-00, Oxford University Computing Laboratory, 2000.

[8] Y. Chen and J. W. Sanders, Logic of global synchrony, *Technical Report* PRG-RR-01-01, Oxford University Computing Laboratory, 2001.

[9] E. W. Dijkstra, Guarded commands, non-determinacy and the formal derivation of programs, *Communications of the ACM*, **18**: 453-457, 1975.

[10] J. He, K. Seidel and A. K. McIver, Probabilistic models for the guarded command language. *Science of Computer Programming* **28**:171-192, 1997.

[11] E. C. R. Hehner, Predicative Programming I, II *Communications of ACM*, **27**(2): 134-151, 1984.

[12] C. A. R. Hoare, Programs are predicates, ed. J. C. Shepherdson and C. A. R. Hoare, *Mathematical Logic and Programming Languages* pp141-155, Prentice Hall, 1985.

[13] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall series in Computer Science, 1985.

[14] C. A. R. Hoare and J. He, The weakest prespecification I,II. *Fundamenta Informatica*, **9**: 51-84, 217-252, 1986.

[15] C. A. R. Hoare et al. Laws of programming, *Communications of the ACM*, **30**(8): 672-686, 1987.

[16] C. A. R. Hoare and J. He, *Unifying Theories of Programming*, Prentice Hall series in Computer Science, 1998.

[17] R. Milner, *Communication and Concurrency*, Prentice Hall series in Computer Science, 1989.

[18] L. Lamport, A temporal logic of actions, *ACM Transctions on Programming Languages and Systems*, **16**(3): 872-923, 1994.

[19] L. Lai and J. W. Sanders, A weakest-environment calculus for communicating processes, *Proc. 4th Nordic Transputer Conference: Parallel Programming and Applications*, ed. P. Fritzon and L. Finmo, 381-395, IOS Press, Ohmsha, 1995.

[20] D. S. Lecomber, *Methods of* BSP *Programming*, Oxford University Computing Laboratory DPhil. thesis, 1998.

[21] R. D. Maddux, Fundamental study relation-algebraic semantics, *Theoretical Computer Science*, **160**: 1-85, 1996.

[22] J. L. Peterson, *Petri Net Theory and Modeling of Systems*, Prentice Hall series in Computer Science, 1981.

[23] L. G. Valiant, A bridging model for parallel computation, *Communications of the ACM*, **33**(8): 103-111, 1990.

[24] J. Woodcock and J. Davis, *Using Z : specification, refinement, and proof*, Prentice Hall series in Computer Science, 1991.