# A denotational semantic theory of concurrent systems

Jayadev Misra

August 27, 2014

## Contents

## 1 Introduction

This paper proposes a general denotational semantic theory suitable for most concurrent systems. It is based on well-known concepts of events, traces and specifications of systems as sets of traces.

A concurrent system consists of a number of components that are combined using the *combinators* of a specific programming language. A specification of a component is a set of traces. A *transformer* combines the specifications of the components to yield the specification of a system; thus, each combinator of a programming language is modeled by a transformer. The two most significant ideas in this paper are *smooth* and *bismooth* transformers that correspond to monotonic and continuous functions in denotational theory [11]. These transformers can model various features of concurrent systems such as, concurrent interactions with memory and objects, independent as well as causally dependent threads, unbounded non-determinism, shared resource, deadlock, fairness, divergence and recursion.

Treatment of Recursion, Section 4, requires us to introduce *bismooth* transformer, the counterpart of a continuous function that preserves the limits of chains (upward-closure) as well as the prefixes of traces (downward-closure). We develop a version of the well-known fixed point theorem [7, 11] that shows that first computing a simple fixed point and then taking its limit is appropriate for bismooth transformers, see Section 4.4 (page 26). Transformers that encode fairness are smooth (monotonic) but not bismooth (continuous); so this theorem does not apply . We generalize the least fixed point theorem, to *min-max* fixed point theorem, for smooth transformers; see Section 4.5 (page 27) and Section 4.6 (page 28).

Monotonic and continuous functions in denotational semantics operate on elements of any complete partial order without any pre-assumed structure. Even though smooth and bismooth transformers are the counterparts of monotonic and continuous functions, they operate on specifications which have structure as sets of traces. We exploit this structural information to obtain strong results about various classes of transformers and fixed points.

We do not to develop the semantics of a specific programming language but of transformers that are of general applicability in all conceivable concurrent systems. Features of specific programming languages can be treated by combining a few elementary transformers, as we demonstrate in the example below.

**A motivating example** Let $\oplus$ be a 3-way combinator so that in $\oplus(A, B, C)$ synchronization of the executions $A$ and $B$ initiates the execution of $C$. Operationally, $A$ and $B$ are parent threads that execute concurrently at start. Child thread $C$ starts executing only when the parents synchronize, by $A$ enagaging in event $e$ and $B$ in $\overline{e}$. In case both events occur, they have completed a "rendezvous", $C$ is started and $A$ and $B$ resume execution. Neither $e$ nor $\overline{e}$ is shown explicitly as occurring in the execution in case of a rendezvous.

It is possible that a synchronization may never be completed even though one of $A$ and $B$, say $A$, has engaged in its synchronization event $e$. In that case, $A$ remains waiting to synchronize and $C$ is never started, though $B$ may continue to execute forever or halt without synchronization.

We define a transformer $\oplus'$, corresponding to the combinator $\oplus$, that transforms the *specifications* of $A$, $B$ and $C$ to yield the specification of $\oplus(A, B, C)$. The definition of $\oplus'$ uses a few transformers described in this paper.

Let the specifications of $A$, $B$ and $C$ be $p$, $q$ and $r$, respectively. Introduce $C'$ that behaves as $C$ but indicates the start of its execution by a specific event $a$; event $a$ does not occur in $p$, $q$ or $r$. The specification of $C'$ is $cons(a, r)$ that appends $a$ as the first event to every trace in $r$; see the definition of *cons* in Section 3.3.5 (page 11). The execution of $\oplus(A, B, C')$ interleaves their individual executions arbitrarily, subject to the constraint that the events $e$, $\overline{e}$ and $a$ be synchronized. The interleaved executions of $A$, $B$ and $C'$ is given by their unfair merge, written as $p \mid q \mid cons(a, r)$; see Section 3.3.11 (page 14). The synchronization of $e$, $\overline{e}$ and $a$ is written using a transformer, called *rendezvous*, that introduces a new event $\tau$ to indicate the simultaneous occurrences of $e$,

$\overline{e}$ and $a$; see Section 3.3.14 (page 17). Finally, event $\tau$ is removed from the specification, using transformer *drop*; see Section 3.3.4 (page 11). Thus,

$$\oplus'(p, q, r) = drop(\{\tau\}, rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r))))$$

We can now assert certain properties of $\oplus'$. For example, that it is bismooth, because all the transformers in its definition are bismooth and composition of bismooth transformers is bismooth.

## 2 Basic Concepts

A *trace* represents one possible execution of a component. The *specification* of a component is a set of prefix-closed *traces*. We define these concepts and explore their properties in this section.

### 2.1 Event and Trace

#### 2.1.1 Event

Events are uninterpreted symbols drawn from an event alphabet. The choice of events for a component constitutes a design decision about the granularity at which we may wish to examine the component. For the systems that we consider in this paper, events could be of many different types including, for instance: input and output, binding of a parameter to a value, calling a shared resource for read/write access, receiving a response from a resource, locking and unlocking of a resource, allocation and disposal of storage, or publishing a value as a result of a computation. Each event denotes a single occurrence of some event type in an execution so all events in an execution are distinct. The semantic theory makes no assumption about the meanings of events.

#### 2.1.2 Trace

A *trace* is the formal counterpart of partial or complete execution of a program. A trace includes a sequence of events, the events occurring in the execution, and the state at the end of the execution if it is finite; the state is called *status* in this paper[1]. An execution that has halted, i.e., one that can engage in no further event, has status $H$. A finite execution that is waiting for an event to happen or waiting to halt has status $W$. An infinite execution has status $D$, representing divergence. A finite execution may also have status $D$; this represents an infinite execution that has only a finite prefix of visible events; see Section 2.4 (page 7).

A trace is written in the form $y[m]$ where $y$ is the *status* from $\{H,\ W,\ D\}$, and $m$ is a finite or infinite sequence of distinct events. If $y$ is $H$ or $W$ then $m$ is finite.

[1]We use the term *status* to distinguish the state of execution from the states of other mutable objects in the system.

### 2.1.3 Example

Consider a component that has the following behavior. It tosses a coin repeatedly until the coin lands heads. Then it halts. Let $hd$ and $tl$ denote the events of coin landing heads and tails, respectively, and $tl^i$ a sequence of length $i$ of $tl$ events. Then any finite execution is represented by, for some $i \geq 0$, either (1) $W[tl^i]$, (2) $W[tl^i\ hd]$, or (3) $H[tl^i\ hd]$. If the coin is fair we expect it to land heads eventually; so, these are the only traces of the component. If the coin is unfair, it is possible to have an infinite sequence of tails, and the corresponding trace is $D[tl^\omega]$. If the coin toss events are invisible, then the only traces in an external spec for fair coin are $W[\,]$ and $H[\,]$, and for unfair coin are $W[\,]$, $H[\,]$ and $D[\,]$. Thus, with an unfair coin an external observer can assert only that this component may eventually halt or may compute forever.

The component described in this example does not interact with any other component. To see interaction, suppose the component does not actually toss the coin but requests another component to do so and communicate the result to it. Let *toss* be a request for a toss, and *rcvhd* and *rcvtl* are the events corresponding to the responses received when the toss lands heads and tails, respectively; assume that a response is guaranteed. A trace of the component with a fair coin is, for some $i \geq 0$, either (1) $W[(toss\ rcvtl)^i]$, (2) $W[(toss\ rcvtl)^i\ toss]$, (3) $W[(toss\ rcvtl)^i\ toss\ rcvhd]$, or (4) $H[(toss\ rcvtl)^i\ toss\ rcvhd]$. An external observer can assert eventual termination, because there is no external event for which the program may wait forever. With an unfair coin there is an additional trace $D[(toss\ rcvtl)^\omega]$, and termination can not be asserted.

**Tuples of traces** In dealing with programs that contain several components, a transformer maps each *tuple* of traces, with one trace from each component, to a set of possible traces of the program. In most contexts the distinction between a trace and a tuple of traces is immaterial, so we use the term "trace" to denote a single trace or a finite tuple of traces, the tuple size depending on the context. A tuple of traces is finite if each component trace is finite.

**Traceset** A *traceset* is a *non-empty* set of traces. A finitary traceset is one in which each trace is finite. Tracesets are partially ordered by subset order.

## 2.2 Prefix order over traces

Informally, trace $s$ is a prefix of $t$ when the execution corresponding to $s$ can possibly be extended to that for $t$. For sequences $m$ and $n$, let $m \sqsubseteq n$ denote that $m$ is a prefix of $n$. Impose a partial order $\leq$ over the status values as follows: $W \leq H$ and $W \leq D$.

Trace $y[m]$ is a *prefix* of $z[n]$ ($z[n]$ an *extension* of $y[m]$) if $y \leq z$ and $m \sqsubseteq n$. And, $y[m]$ is a *proper prefix* of $z[n]$, if $y[m] \leq z[n]$ and $y[m] \neq z[n]$. So, a trace with status $H$ or $D$ has no extension. An infinite trace is a prefix only of itself. And $W[\,] \leq y[m]$ for every trace $y[m]$. Observe that $W[m] < H[m]$ even though the event sequences of both traces are identical; this denotes that $W[m]$ has to causally precede $H[m]$ in any execution.

For tuples of traces define one tuple as a prefix of another if each entry in the former tuple is a prefix of the corresponding entry in the latter. And $(s_0, s_1, \cdots, s_k) < (t_0, t_1, \cdots, t_k)$ if $s_i \leq t_i$ for each $i$ and $s_j < t_j$ for some $j$.

**Properties of prefix order** The following properties are easy to prove.

1. Prefix order, $\leq$, is a partial order over traces.
2. The inverse of proper prefix order, $>$, is a well-founded order over traces.
3. The set of prefixes of a trace are totally ordered.

**An Induction Principle over traces** The inverse of proper prefix order, $>$, is a well-founded order even in the presence of infinite traces. This allows us to formulate the following induction principle. Let $P$ be a predicate over traces, both finite and infinite.

If for all $t$, $(\forall s : s < t : P(s)) \;\Rightarrow\; P(t)$,
then $P(t)$ holds for all traces $t$.

## 2.3 Prefix Closure

The *prefix-closure*, also called *downward-closure*, of trace $t$ is denoted by $t_*$; it is the set of all its prefixes of $t$. For a traceset $p$, $p_*$ is the set of prefixes of all traces of $p$. That is,

$$t_* = \{s \mid s \leq t\} \text{ and } p_* = \cup\{t_* \mid t \in p\}.$$

It follows that for traces $s$ and $t$, $(s,t)_* = s_* \times t_*$.

**Finite Prefix-Closure** Denote the set of *finite* prefixes of trace $t$ by $t_{*'}$. Define $p_{*'}$ for traceset $p$ analogously. Note that an infinite trace $t$ is not in $t_{*'}$, though $t \in t_*$.

**Notational Conventions**

1. Prefix-closure and finite prefix-closure operators have the highest binding power among all operators.
2. Prefix closure and finite prefix-closure apply to event sequences, not just traces and tracesets.
3. Write $C_*(p)$ for $(C(p))_*$ for any $p$ in any context $C$.

4. (singletons and sets) A singleton trace may appear wherever a traceset is expected to appear. That is, if $C(p)$ is a valid expression for any traceset $p$, so is $C(t)$ for a trace $t$, and it denotes $C(\{t\})$.

   Conversely, if $C(t)$ is a valid expression for any trace $t$, so is $C(p)$ for any traceset $p$, and it denotes $\cup_{t \in p} C(t)$.

Thus, $W[m_*]$ is a shorthand for $\{W[k] \mid k \in m_*\}$. And, $W_*[m] = (W[m])_* = \{s \mid s \leq W[m]\}$.

**Elementary Properties of Prefix-Closure** Below $p$ and $q$ are tracesets, and $t$ any trace. The following properties are easy to show. Closure expansion, item (6), is used extensively in subsequent proofs.

1. Prefix-closure is algebraic closure, i.e., for tracesets $p$ and $q$,

   (a) (extensive) $p \subseteq p_*$

   (b) (monotonic) $p \subseteq q \;\Rightarrow\; p_* \subseteq q_*$

   (c) (idempotent) $(p_*)_* = p_*$.

2. Finite prefix-closure of tracesets is monotonic and idempotent. Extensive property does not hold for the traceset $\{t\}$ where $t$ is an infinite trace.

3. $(t_*)_{*'} = (t_{*'})_* = t_{*'}$.

4. (Closure expansion) For any trace $z[m]$, $z_*[m] = \{z[m]\} \cup W[m_{*'}]$.

5. (closure distributes over set union) For a family $F$ of tracesets, $F$ possibly infinite, $(\cup_{p \in F}(p))_* = (\cup_{p \in F}(p_*))$.

6. (closure distributes over Cartesian product) $(p \times q)_* = p_* \times q_*$.

## 2.4 Specification

Informally, a specification of a component, henceforth abbreviated as *spec*, is a set of traces, where each trace corresponds to an execution of the component in *some* environment. Different traces may correspond to executions in different environments. Properties of a component may be deduced from its spec, such as that its publications are monotonic in value (a safety property), every execution eventually halts (a progress property), or that the component's execution may deadlock (the spec includes a trace $W[m]$ that has no extension).

**Spec** A *spec* is a prefix-closed traceset. A *finitary spec* is a spec consisting of finite traces.

Note that a spec of $n$-tuples includes the bottom trace, $(W[\,], W[\,], \cdots, W[\,])$, consisting of $n$ individual empty traces.

### 2.4.1 Properties of Specs

The proofs of the following properties are elementary.

1. For any traceset $p$, $p_*$ and $p_{*'}$ are both specs.
2. Union of a finite or infinite family of specs is a spec.
3. Intersection of a finite or infinite family of specs is a spec.
4. Cartesian product of a pair of specs is a spec.

Consider the coin toss example of Section 2.1.3 (page 5). With a fair coin we expect the spec to be $H_*[tl^i\ hd]$, and for an unfair coin to be $H_*[tl^i\ hd] \cup \{D[tl^\omega]\}$.

### 2.4.2 Chains and their limits

A *chain* is a finitary spec whose elements are totally ordered under $\leq$. A chain may be finite or infinite. For any trace $t$ the set of its finite prefixes, $t_{*'}$, is a chain.

The limit of chain $c$, written as $lim(c)$, is the least upper bound of the traces in $c$ with respect to the $\leq$ ordering. For a finite chain $c$, $lim(c)$ is the longest trace in $c$. For an infinite chain $c$, $lim(c)$ is the unique infinite trace such that every trace in $c$ is its prefix. Note that $lim(c)$ does not belong to $c$ for infinite $c$ because $c$ consists of finite traces only. Notationally, use $lim(c)$ as a trace and also as a singleton traceset.

Define the limit of a finite tuple of chains as the tuple of limits of the corresponding chains. That is,

$$lim(c_0, c_1, \cdots c_n) = (lim(c_0), lim(c_1), \cdots lim(c_n))$$

### 2.4.3 Complete lattice of specs

The least upper bound of a set of specs is their union, and the greatest lower bound is the intersection. Thus, specs form a complete lattice under subset order, where $\bot = W[\,]$ and $\top$ is the union of all specs.

## 3 Transformer

Any component of a system is either a *primitive* component or a *structured* component. A primitive component is defined by its spec. A structured component consists of one or more subcomponents that are combined using the *combinators* of the language. A spec transformer, or simply a *transformer*, corresponding to each combinator is a function mapping the Cartesian product of the specs of the subcomponents to the spec of the structured component. The number of subcomponents, therefore the length of the tuples in the argument of the transformer, is the *arity* of the transformer. A language semantic thus consists of the specs of the primitive components and the transformers corresponding to each combinator. For the moment assume that the domain of a transformer is the set of all traces. We show how to restrict the domain of a transformer in Section 3.2.2 (page 10).

**Convention** We develop the theory for transformers of arity 1, a transformer that maps a spec to a spec. Generalizations for other arities are straightforward. Examples of transformers of higher arity appear in Section 3.3.2 (page 11), Section 3.3.11 (page 14), Section 3.3.12 (page 15) and Section 3.3.15 (page 17). For a transformer of arity 2 we adopt infix notation, as in $p \oplus q$.

We restrict ourselves to a class of transformers, called *smooth*. Smooth transformers correspond to monotonic functions in denotational semantic theory. A subset of smooth transformers, called *bismooth*, correspond to continuous functions. We develop the theory of smooth transformers in this section and bismooth transformers in Section 4.3 (page 24).

## 3.1 Trace-wise transformer

A trace-wise transformer is a total function from traces to tracesets. Following the notational convention from page 6, a trace-wise transformer $f$ applied to a traceset $p$ is defined to be: $f(p) = \cup\{f(t) \mid t \in p\}$. For trace-wise combinator $\oplus$ over a pair of specs, $p \oplus q = \cup\{s \oplus t \mid s \in p,\ t \in q\}$.

Any transformer maps a spec to a spec. We restrict ourselves to trace-wise transformers in this paper because a language combinator can combine only individual executions of its components. Non-determinism issues represented by mapping a trace to a traceset, every trace of the latter corresponds to a possible execution. The size of the traceset is arbitrary thus allowing unbounded non-determinism.

### 3.1.1 Properties of trace-wise transformers

The following properties follow from the definition of trace-wise transformers.

1. A trace-wise transformer distributes over union (possibly infinite union) of tracesets. That is, given a family $F$ of tracesets,

$$(\cup_{p \in F} f(p)) = f(\cup_{p \in F}(p))$$

2. Composition of trace-wise transformers is a trace-wise transformer.

3. (Monotonicity) For trace-wise $f$ and tracesets $p$ and $q$,
$p \subseteq q \;\Rightarrow\; f(p) \subseteq f(q)$.

A trace-wise transformer may not transform a spec to a spec, i.e., the resulting traceset may not be prefix-closed (consider a transformer that maps every trace to $W[a]$ where $a$ is some event; the resulting traceset does not include $W[\,]$, hence, is not a spec). The smoothness condition, described below, guarantees this property.

## 3.2 Smooth Transformer

A transformer $f$ is *smooth* if and only if for any traceset $p$

$f_*(p) = f(p_*)$, where $f_*(p)$ stands for $(f(p))_*$.

A transformer $f$ is *finitely smooth* if and only if for any finitary traceset $p$, $f_*(p) = f(p_*)$.

### 3.2.1 Properties of smooth transformers

1. A transformer $f$ is smooth if and only if it preserves prefix-closure over individual traces, i.e., $f_*(t) = f(t_*)$, for every trace $t$.

   Proof: See Proposition 1 (page 31) in the Appendix.

2. A transformer is smooth if and only if it maps specs to specs.

   Proof: See Proposition 2 (page 31) in the Appendix.

3. Composition of smooth transformers is smooth.

   Proof: See Proposition 3 (page 32) in the Appendix.

**Terminology and Notation** Henceforth, "transformer" stands for "trace-wise transformer" in this paper. For a binary smooth transformer $\oplus$ written in infix style, $(p \oplus q)_* = p_* \oplus q_*$, for tracesets $p$ and $q$.

### 3.2.2 Domain of a transformer

We have so far assumed that every transformer is defined for all traces. In many cases a transformer $f$ can meaningfully be defined only over some domain $dom(f)$; we assume that $dom(f)$ is a spec. We show how to extend the domain of a transformer while retaining its essential properties. Specifically, we define transformer $g$ over all traces that induces the same mapping over $dom(f)$ as $f$ and retains smoothness and bismoothness.

For any $t$ in $dom(f)$ let $g(t) = f(t)$. For $t \notin dom(f)$ and finite $t$, let $g(t) = \cup\{f(s) \mid s \leq t \text{ and } s \in dom(f)\}$. For $t \notin dom(f)$ and infinite $t$, let $g(t) = lim(g(t_{*'}))$, where $lim$ is defined in Section 2.4.2 (page 8). It can be shown that if $f$ is smooth over the traces in $dom(f)$ then so is $g$ over all traces, and if $f$ is bismooth over any spec in $dom(f)$ then so is $g$ over all specs.

Note: For $t \notin dom(f)$ and finite $t$, alternately let $g(t) = f(s)$ where $s$ is the longest prefix of $t$ in $dom(f)$.

## 3.3 Some Elementary Smooth Transformers

In this section, we show a number of smooth transformers that are of general utility. Transformer $g$ with arguments is written as $g(args, t)$ where $args$ is a set of parameters and $t$ a trace. Here, $g$ represents a family of transformers, one transformer for each value of $args$. For a specific value of $args$ we abbreviate $g(args, t)$ to $f(t)$, and then prove the smoothness of $f$. We note that the identity transformer, $id(t) = t$ for all traces $t$, is smooth.

### 3.3.1 Status map

This is a family of transformers each member of which may change the status of a trace but not its event sequence. Applying $statusmap(y[m])$, a generic member of the family, yields $y'[m]$ where $y'$ may differ from $y$ only if $y = H$, or $y = D$ and $m$ is finite; thus, $statusmap(y[m]) = y[m]$, if $y = W$ or $m$ is infinite. We show that every transformer in $statusmap$ is smooth. See Proposition 4 (page 32) in the Appendix.

### 3.3.2 Choice

The choice transformer, *or*, corresponds to a non-deterministic choice between two components to execute. For components $f$ and $g$ with specs $p$ and $q$, $f \ or \ g$ has the spec $p \cup q$. As a trace-wise transformer:

$$s \ or \ t = \{s, t\}$$

We show that *or* is smooth in Proposition 5 (page 33) in the Appendix.

### 3.3.3 hide

Transformer *hide* is parameterized by a set of events $E$, which may be finite or infinite; $hide(E, t)$ is the trace obtained after removing all events from $t$ that also occur in $E$. Application of *hide* may remove an unbounded, and possibly infinite, number of events from a trace. For example, $hide(\{a\}, D[a^\omega])$ results in $D[\,]$.

We show that *hide* is smooth for any $E$ in Proposition 6 (page 33) in the Appendix.

### 3.3.4 drop

Transformer *drop* is same as *hide* except that in $drop(E, t)$ (1) the event set $E$ is finite, and (2) only the first occurrence, if any, of an event from $E$ is removed from $t$, but subsequent occurrences are retained. The proof that *drop* is smooth is similar to the proof for *hide*. The reason we treat *drop* separately is that *drop* is bismooth —see Section 4.3.4 (page 26)— whereas *hide* is not. This property permits *drop*, but not *hide*, to be freely used in recursive equations.

### 3.3.5 cons

Append a specific event $a$ as the first event of every trace. To ensure that a spec is transformed to a spec, $cons(a, W[\,])$ includes $W[\,]$.

$cons(a, W[\,]) = \{W[\,], W[a]\}$
$cons(a, y[m]) = \{y[am]\}$

We show that *cons* is smooth in Proposition 7 (page 33) in the Appendix.

### 3.3.6 Filter

A class of transformers, called *filter*, is essential for most applications of this theory. A filter can be used to model interactions among components by rejecting the traces that do not implement acceptable interactions, as in accesses to shared resources. A filter can also model rendezvous-style interactions and fairness constraints.

Associated with each filter is a predicate $b$ over traces such that:

F1. $b(W[\,])$ holds, and

F2. If $b(t)$ holds then $b(s)$ holds for all proper prefixes $s$ of $t$, i.e., writing $b(t_{*'})$ for the conjunction of $b(s)$ over all finite prefixes $s$ of $t$: $b(t) \Rightarrow b(t_{*'})$.

Filter $f$ corresponding to predicate $b$ *accepts* $t$ iff $b(t)$ holds and *rejects* it otherwise. Thus, if a filter accepts a trace it accepts all prefixes of that trace; equivalently, if it rejects a trace, it rejects all extensions of that trace. Since $b(W[\,])$ holds, not all traces are rejected. A filter applied to a spec retains only its acceptable traces. In a proof theory (which we do not develop here) we would use a predicate to describe the traces in a spec. Applying a filter to a spec amounts to conjoining the filter predicate to eliminate the unacceptable traces.

The natural definition of transformer $f$ corresponding to filter predicate $b$ is $f(t) = \{t\}$ if $b(t)$ and $\{\}$ otherwise. This definition violates the requirement that $f(t)$ be a traceset, a non-empty set of traces, for all $t$. So, we propose:

$$f(t) = \{s \mid s \le t \text{ and } b(s)\}$$

Transformer $f$ is smooth; see Proposition 15 (page 38) in the Appendix.

Observe that for filter predicates $b$ and $b'$, $b \wedge b'$ and $b \vee b'$ are also filter predicates. If transformers $g$ and $g'$ implement $b$ and $b'$ respectively, then $g \circ g'$ implements $b \wedge b'$ and $g(t) \cup g'(t)$, for any trace $t$, implements the disjunction of the filters. Any filter transformer is idempotent, and it distributes over union and intersection of specs. The following identity is used in the min-max fixed point theorem, Section 4.5 (page 27). For filter $g$ and specs $p$ and $q$,

$$g(p \cap q) = g(p) \cap g(q) = g(p) \cap q$$

**Continuous vs. Discontinuous Filter** We distinguish between two kinds of filters, *continuous* and *discontinuous*, depending on the value of $b(t)$ for infinite $t$. A discontinuous filter models fairness wherein an infinite trace may be rejected even though all its finite prefixes are accepted. A continuous filter rejects an infinite trace only if some finite prefix of it is also rejected. Conversely, a continuous filter accepts an infinite trace if all its finite prefixes are accepted.

Both types of filter predicates obey the conditions (F1) and (F2) given earlier. Additionally, a continuous filter predicate $b$ satisfies the stronger condition (F2') below in place of (F2):

F2'. $b(t) \equiv b(t_{*'})$, for every trace $t$.

Note that (F2) and (F2') are equivalent for finite $t$. It is only for infinite $t$ that (F2') imposes the additional constraint: $b(t_{*'}) \Rightarrow b(t)$.

For a continuous filter $f$, we can let $f(t)$ be the the longest prefix of $t$ for which $b$ holds. This is defined for finite $t$ because $b(W[\,])$ holds, and for infinite $t$ because the longest prefix is $t$ if $b(t)$ holds and some finite prefix of $t$ if $\neg b(t)$ holds.

Continuous filters are always bismooth, discontinuous filters are not; see Section 4.3.4 (page 26).

Filters are some of the most useful transformers. The following sections list special cases of filters that arise in concurrent programming.

**Partitioning a filter** Any filter can be written as a composition of two filters one of which is continuous and the other rejects only infinite traces. That is, a filter $f$ can be written as $f_{inf} \circ f_{fin}$ where (1) $f_{inf}$ rejects trace $t$ only if $t$ is infinite, and $f$ rejects $t$ though it accepts all finite prefixes of $t$, and (2) $f_{fin}$ rejects all other traces, finite and infinite, that $f$ rejects. It is possible that neither $f_{inf}$ nor $f_{fin}$ rejects any trace. Clearly, $f = f_{inf} \circ f_{fin}$. Further, $f_{fin}$ is a continuous filter because whenever it rejects an infinite trace it also rejects a finite prefix of it. And, if $f_{inf}$ rejects any trace, it is a discontinuous filter.

### 3.3.7 Restrict by inclusion of events

Reject a trace if it contains a specific event $a$, or, more generally, an event from a specified set $E$. This is a filter because (1) it accepts $W[\,]$, and (2) if it accepts a trace, it accepts all its prefixes. The filter is continuous.

The converse of this rejection criterion is *not* smooth: accept a trace only if it is $W[\,]$ or contains a specific event $a$. Then any trace that has $a$ as its last event is accepted but all its prefixes except $W[\,]$ are rejected. Therefore, it may transform a spec to a traceset that is not prefix-closed.

### 3.3.8 Restrict by exclusion of events

Accept a trace only if it is $W[\,]$ or its *first* event is drawn from a specified set of events. This condition defines a filter predicate $b$ because: (1) $b(W[\,])$ holds, and (2) if $b(t)$ holds, it holds for all prefixes of $t$. The filter is continuous. The requirement that the specified event be the first one in the event sequence is crucial; without this requirement the transformer is not smooth.

The acceptance criterion here is stronger than a typical filter: whenever a trace is accepted, all its extensions are also accepted.

### 3.3.9 Restrict by precedence relation

Let $R$ be a binary relation over events. Define a transformer that accepts trace $t$ iff for every $(e, e')$ in $R$, if $e'$ is in $t$ then $e$ is also in $t$ and $e$ precedes $e'$. Thus, an acceptable trace is one that either includes (1) none of $e$ and $e'$, (2) just $e$, or (3) both $e$ and $e'$ with $e$ preceding $e'$. It is easy to see that $W[\,]$ is accepted and the prefix of an acceptable trace is acceptable. Further, this transformer is a continuous filter.

### 3.3.10 atom

Atomicity is a fundamental notion in concurrent programming, particularly in the theory of transactions. Roughly, trace $t$ is atomic with respect to a specified set of events if all the specified events occur contiguously in some order in $t$. We propose a more general definition that is useful in defining other transformers.

A *pattern alphabet* is a finite subset of the event alphabet. A *pattern* is a finite string over the pattern alphabet. Let $P$ be a finite set of patterns. Trace $t$ is *atomic* with respect to $P$ if the event sequence in $t$ can be written uniquely as a sequence of patterns from $P$ interspersed with events outside the pattern alphabet, optionally followed by a prefix of some pattern if $t$ is finite. Predicate $atom(P, t)$, where $P$ is a finite set of patterns and $t$ a trace, holds iff $t$ is atomic with respect to $P$.

It is easy to see that *atom* is a filter predicate, because $W[\,]$ is accepted and if $t$ is accepted then so are all its prefixes. Additionally, *atom* defines a continuous filter because if an infinite trace $t$ is rejected then some finite prefix of it is not atomic with respect to $P$.

### 3.3.11 Unfair merge

One of the most important transformers, that models concurrent executions of components, is *merge*. It interleaves the events of two traces arbitrarily yielding a traceset from a pair of traces. Besides interleaving the events, merge also computes the status of the interleaved trace based on those of the given traces. Assume that the events in the traces to be merged are distinct.

There are two forms of interleavings, *unfair* and *fair*, of event sequences $m$ and $n$. The distinction is significant only when one or both of $m$ and $n$ are infinite. If each interleaving includes all elements of $m$ and $n$ then it is *fair*; we treat fair merge in Section 3.3.12 (page 15). An unfair interleaving may include only a finite prefix of $n$ for infinite $m$, and analogously for infinite $n$.

**Properties of unfair interleaving** Define unfair interleaving of $m$ and $n$, $m \otimes n$, by the following program (written in a functional programming style):

$$
\begin{aligned}
&[\,]\otimes n = n\\
&m\otimes[\,] = m\\
&(a:m)\otimes(b:n) = (a:(m\otimes(b:n)))\cup(b:((a:m)\otimes n))
\end{aligned}
$$

Using fixed point induction it can be shown that $\otimes$ is symmetric. It can also be shown that it is monotonic in both arguments, so $m\otimes n\subseteq m'\otimes n$ and $m\otimes n\subseteq m\otimes n'$, where $m\subseteq m'$ and $n\subseteq n'$. Further,

$$(m\otimes n)_* = m_*\otimes n_* \qquad (\otimes \text{ distributes over prefixes})$$

**Transformer for unfair merge** Unfair merge of two traces applies unfair interleaving to their event sequences. Also, it applies a symmetric binary operation $\cap$ over their status values: $H\cap y = y$ and $W\cap W = W$. Define unfair merge transformer, $\mid$ , as follows, where both $y$ and $z$ are from $\{H,\ W\}$.

$$
\begin{aligned}
y[m]\mid z[n] &= (y\cap z)(m\otimes n)\\
D[m]\mid z[n] &= D[m\otimes n_*]\\
D[m]\mid D[n] &= D[m\otimes n_*]\cup D[m_*\otimes n]
\end{aligned}
$$

Observe that $m$ and $n$ may be finite or infinite in $D[m]$ and $D[n]$ above. Note that $z[n]\mid D[m] = D[m]\mid z[n]$, and it is not shown explicitly below.

The intuition behind this definition is as follows. Expression $y[m]\mid z[n]$ denotes the concurrent execution of two executions, one corresponding to $y[m]$ and the other to $z[n]$. Both executions are finite and if either fails to halt then the concurrent execution does not halt either, as given by the status $(y\cap z)$. The event sequence in $y[m]\mid z[n]$ is an interleaving of $m$ and $n$, which justifies the result expression $(y\cap z)(m\otimes n)$.

Next, consider infinite executions defined by the next two cases. In $D[m]\mid z[n]$, $D[m]$ denotes an infinite execution $D[m']$ where $m$ is the sequence of visible events in $m'$; thus, $m$ may be finite. The resulting concurrent execution is infinite, so its status is $D$. Any concurrent execution executes a prefix of $z[n]$ with all of $D[m']$; so the event sequences in all such executions are given by $m'\otimes n$. Since only the events of $m$ are retained from $m'$, the resulting expression is $D[m\otimes n_*]$. Similar remarks apply for $D[m]\mid D[n]$, because any execution may use a prefix of the event sequence of one of $D[m]$ or $D[n]$ and all events of the other.

It can be shown from the above definition that unfair merge is commutative, associative and $H[\ ]$ is its zero. We show that unfair merge is smooth in Proposition 8 (page 34) in the Appendix.

### 3.3.12 Fair merge

Fair merge is based on fair interleaving, which we denote by $\otimes'$:

$$m\otimes' n = \{x \mid x\in m\otimes n,\ x \text{ contains } m \text{ and } n \text{ as subsequences}\}$$

Note that if $m$ is infinite and $n$ non-empty, then $m \in m \otimes n$ and $m \notin m \otimes' n$.

Extend the definition of $\cap$ to apply to all status values $\{H, W, D\}$ as follows. Recall that $\cap$ is symmetric. For any status value $y$

$$H \cap y = y,\ W \cap W = W \text{ and } D \cap y = D$$

Define fair merge transformer, $|'$ , of two argument traces $y[m]$ and $z[n]$ for $y$ and $z$ from $\{H, W, D\}$, and finite or infinite $m$ and $n$.

$$y[m] \ |' \ z[n] = (y \cap z)(m \otimes' n)$$

The proof of smoothness of fair merge can be developed in a manner similar to unfair merge. There is a much simpler alternative proof. Observe that fair merge of $y[m]$ and $z[n]$ is same as their unfair merge followed by application of a filter that removes every infinite trace $D[k]$ from $y[m] \mid z[n]$ where $k \notin m \otimes' n$. Both unfair merge and filter are smooth; so, their composition, fair merge, is also smooth.

### 3.3.13 replace

We consider a general version of substitution of a sequence of events by a single event. A *source alphabet* and a *target alphabet* are disjoint finite subsets of the event alphabet. A *replacement pair* is of the form $(\sigma, \tau)$ where $\sigma$, called the *source*, is a finite string over the source alphabet, and $\tau$, called the *target*, is a single symbol from the target alphabet.

Let $R$ be a finite set of replacement pairs. A source may occur multiple times in $R$ with different targets, and similarly, a target may have multiple occurrences in $R$ with different sources. Transformer *replace* substitutes occurrences of a source by all corresponding targets in an event sequence. The effect of $replace(R, t)$ is to (1) accept $t$ if $t$ is atomic with respect to the sources, see Section 3.3.10 (page 14) and $t$ contain no symbol from the target alphabet, and (2) if $t$ is accepted, replace occurrence of every source by *all* corresponding targets to obtain a set of traces, and (3) then replace occurrence of any proper prefix of a source by the empty string. The situation in (3) arises because the prefix of an atomic trace may contain a prefix of a source as its suffix. The domain of this transformer can be extended to all traces using domain extension described in Section 3.2.2 (page 10).

Henceforth, let $f(t)$ denote $replace(R, t)$ for a specific $R$. The definition of $f$ for finite $t$ is given in clausal form in a functional style, where the clauses are attempted in the given order from top to bottom.

$$\begin{array}{lll} f(y[\sigma']) & = & y[\,], \text{ where } \sigma' \text{ is a proper prefix of a source} \\ f(y[\sigma m]) & = & \cup\{cons(\tau, f(y[m])) \mid (\sigma, \tau) \in R\} \\ f(y[am]) & = & cons(a, f(y[m])),\ a \notin \text{source alphabet} \end{array}$$

Note that the source $\sigma$ is replaced by every target associated with it in $f(y[\sigma m])$.

For infinite $t$, it is easier to specify the transformer using limits from Section 2.4.2 (page 8). Such a definition permits simpler proofs of smoothness and bismoothness: $f(t) = lim(f(t_{*'}))$.

We prove that *replace* is smooth in Proposition 9 (page 35) in the Appendix. It can be shown that the "substitution" transformer, that replaces each event $e$ in a trace by event $h(e)$ where $h$ is a function over the event alphabet, is smooth.

### 3.3.14 Rendezvous

The unfair and fair merge transformers of Section 3.3.11 (page 14) and Section 3.3.12 (page 15) implement independent concurrent processes whose executions can be arbitrarily interleaved. We consider more refined versions of concurrent executions in Section 3.5 (page 18) in which the processes call upon shared resources, and hence, their executions can not be arbitrarily interleaved. Here, we introduce a form of synchronization, called rendezvous in CSP [5] and CCS [10], that ensures that a pair of complementary events $\{e, \overline{e}\}$ from the two processes occur simultaneously. Their simultaneous occurrence is shown by an event $\tau$ in the combined trace that belongs to neither process.

We define *rendezvous* by composing the transformers *atom* and *replace*. First, perform an appropriate merge, fair or unfair, of the specs of the two processes. Then apply transformer *atom* of Section 3.3.10 (page 14) to eliminate the traces in which $\{e, \overline{e}\}$ do not occur contiguously. Next, using transformer *replace* of Section 3.3.13 (page 16), replace all (contiguous) occurrences of $\{e, \overline{e}\}$ by $\tau$, and remove any $e$ or $\overline{e}$ event that occurs by itself. We generalize this scheme slightly by allowing rendezvous to occur with any finite set of events $E$ instead of just two events $\{e, \overline{e}\}$, as follows.

Let $E'$ be the set of strings obtained by permuting the events of $E$ in all possible order. Henceforth, write $rendezvous(E, \tau, t)$, $\tau \notin E$, for the transformer that (1) accepts $t$ provided $t$ is atomic with respect to $E'$, (2) replaces every pattern of $E'$ in trace $t$ by event $\tau$, and then (3) removes any non-empty proper prefix of a pattern of $E'$. Here, $t$ would likely be a trace arising out of the concurrent executions of processes. If required, $\tau$ can be eliminated by applying transformer *drop* of Section 3.3.4 (page 11). Define

$$rendezvous(E, \tau, t) = replace(\{(\sigma, \tau) \mid \sigma \in E'\}, atom(E', t))$$

Since *atom* and *replace* are smooth, so is *rendezvous*.

### 3.3.15 Sequential Composition

Consider a simple form of sequential composition of $f$ and $g$ in which $g$ starts executing only when $f$ halts. The corresponding transformer $\mathbin{;}$ is:

$H[m] \mathbin{;} z[n] = z[mn]$,
$s \mathbin{;} z[n] = s$, otherwise

It can be shown that sequential composition is associative. We show that sequential composition is smooth in Proposition 10 (page 36) in the Appendix.

## 3.4 Fairness

Fairness is a filter that eliminates only certain infinite traces from a spec. For example, a fairness constraint for the coin toss example of Section 2.1.3 (page 5) may specify that the coin is fair so that an infinite sequence of tails is impossible; then, trace $D[tl^{\omega}]$ is inadmissible. A fairness constraint about a strong semaphore may specify that any execution in which a $P$ event on the semaphore remains waiting forever while $V$ events happen infinitely often is inadmissible. In a real-time computation a fairness constraint may specify that an infinite number of events may not occur within a bounded time interval.

Every fairness constraint can be defined by a filter predicate $b$, where $b$ holds for all finite traces and, possibly, some infinite traces. As for any filter predicate, $b(W[\,])$ holds and if $b$ holds for any trace it holds for all its prefixes. For the coin toss example, the filter predicate $b$ holds for every finite trace and every infinite trace that does not have an infinite suffix of either heads or tails (for the example shown, it does not matter if the coin lands heads infinitely often, because the game is terminated after the first landing of a head). Being a filter, fairness is a smooth transformer. We show in Section 4.3.4 (page 26) that fairness is not bismooth.

Fairness can be composed with other transformers. In particular, different forms of fairness may apply to different parts of a program; a fair and an unfair version of the coin toss program may run concurrently, for example, and our theory would yield their combined spec.

## 3.5 Shared Resource

Merge transformers, Section 3.3.11 (page 14) and Section 3.3.12 (page 15), model independent concurrent executions of processes by interleaving the traces of the individual processes. Concurrent executions are rarely independent. For example, trace $s$ of one process includes the event $read(3)$ that reads value 3 from a read/write shared store, trace $t$ of another process includes $write(3)$ for the same store, and the store is local to these two processes. Then $write(3)$ precedes $read(3)$ in the traces for their concurrent execution; any trace in which the events occur in a different order has to be rejected. Further, if $t$ includes $write(5)$ instead of $write(3)$, no trace for the concurrent execution can include $read(3)$.

**Shared resource is a filter** Each resource instance is a filter over an alphabet that denotes the available operations on the resource. Alphabets of different instances of the same resource and of different resources are disjoint. Applied to the merge of traces of individual processes, the filter rejects the traces that violate the semantics of the shared store. For example, for a read/write store that is local to a pair of concurrently executing processes, first the appropriate merge of their traces is constructed, and then a filter applied to ensure that: (1) a value is written to the store before any value is read, and (2) any value that is read is equal to the value last written. Any trace that violates these constraints is rejected. Independent resources are independent filters that may be applied in arbitrary order on a trace.

**Local vs. global resource** Consider concurrently executing processes $A$ and $B$ that include traces $s$ and $t$ in their specs, respectively. Suppose $s$ includes $read(3)$ and $t$ includes $write(5)$, as the only events on a shared read/write store. As we have seen earlier, if the store is local to $A$ and $B$, no trace for the concurrent execution of $A$ and $B$ can include $read(3)$. However, if the store is global, so that other processes may access it, another process may perform $write(3)$. So, a trace for the concurrent execution of $A$ and $B$ may include $read(3)$ for a global store.

It follows from this discussion that each resource has *two* filters corresponding to its local and global behaviors. Suppose processes $A$, $B$ and $C$, whose specs are $p$, $q$ and $r$, respectively, have a local resource. Let $fl$ be the local filter and $fg$ the global filter for the resource. Then the spec for the concurrent execution of $A$, $B$ and $C$ (assuming unfair merge for their concurrent execution) is $fl(fg(p \mid q) \mid r)$. It is easy to see that the global filter for a read/write store accepts all traces, because for any given trace there is a sequence of accesses to the store that validates that trace.

It is possible to develop a more elaborate set of filters for a resource based on access rights that allows different processes to perform different operations on the resource.

**Blocking operations on shared resource** Both filters, local and global, for a read/write store are continuous. In fact, a resource for which all operations are non-blocking induces continuous filters. (Note, however, that for processes that share a read/write store, their concurrent execution is modeled by the fair merge of their specs. A fair merge introduces discontinuity; see Proposition 29 (page 44) in the Appendix.

For a resource with blocking operations, the filter may be continuous or discontinuous. Consider a semaphore that has operations $P$ and $V$ on it, where $P$ is blocking and $V$ non-blocking. It is customary to consider $P$ as consisting of two events, a *request* event, which we denote by $\langle P$ and a *response* event $P\rangle$, where $P\rangle$ is always preceded by the corresponding $\langle P$, though a $\langle P$ may never be followed by a corresponding $P\rangle$.

First, consider a weak semaphore that merely ensures that a request is granted (response sent), whenever the semaphore is available, to *some* waiting process (i.e., any that has an outstanding request for it), though any specific waiting process may never be granted the semaphore. A weak semaphore filter, both local and global, has to reject an infinite trace in which the semaphore is continuously available in an infinite suffix, the suffix contains $\langle P$, but contains no subsequent $P\rangle$. The weak semaphore filter is continuous.

Next, consider a strong semaphore that ensures that each process that requests the semaphore is eventually granted it, provided the semaphore is available infinitely often in an infinite execution. The specification of each process identifies the request and response events by the process identity. The corresponding filter rejects an infinite trace that contains an infinite number of occurrences of $V$, some occurrence of $\langle P_1$ for a specific process numbered 1, but no subsequent $P_1\rangle$. This is a discontinuous filter.

# 4 Treatment of Recursion

The theory developed so far is adequate for programs that include no recursive definition; now, we enhance the theory to treat recursive definitions. Guarded recursion is usually easier to handle. We treat the general case of unguarded recursion, as in solving an equation of the form $x = f(x)$ in spec $x$, for a given transformer $f$. Thus, we will compute the spec of a definition such as

$$\mathit{def}\ \mathit{loop}() = \mathit{loop}()$$

where *loop*(), with no arguments, is defined recursively. As we will see, the spec of this program will not be the bottom spec $\{W[\,]\}$ but $\{D[\,]\}_*$ denoting a divergent computation. This is because we expect each recursive call to engage in an internal event in making the call, so the call entails an infinite computation in which the internal events are invisible.

## 4.1 Classical Treatment of Recursion

We start with the least fixed-point theorem due to Kleene [7], and also in Scott [11], that applies for any continuous function $f$ on a complete partial order (cpo).

**Theorem 1 (Least Fixed-point Theorem)**
Let $f$ be a continuous function on a cpo whose bottom element is $\perp$. The least fixed-point of $f$, $\mathit{lfp}(f)$, is $\mathit{lub}\{f^i(\perp) \mid i \geq 0\}$ where
$f^0(x) = x$, $f^{i+1}(x) = f(f^i(x))$ and $\mathit{lub}$ is the least upper bound of a chain. $\Box$

In applying this theorem in our context, the set of specs form a complete lattice, hence a cpo. Any trace-wise transformer is continuous over specs because given a chain of specs $p_i$, $0 \leq i$, where the least upper bound is union:

$$f(\cup\{p_i \mid i \geq 0\}) = \cup\{f(p_i) \mid i \geq 0)\}.$$

**Corollary 1** The least fixed point of a smooth transformer is a spec.

Proof: It is easily shown by induction on $i$ that for any $i$, $i \geq 0$, $f^i(W[\,])$ is a spec. The union of specs is a spec. So, $\mathit{lfp}(f)$ is a spec, from Theorem 1.

### 4.1.1 Revisiting the coin toss example

As an example of the application of the least fixed-point theorem consider the coin toss example of Section 2.1.3 (page 5). Call the toss program *stutter*. A step of *stutter* either halts the computation, or engages in event *tl* and then calls *stutter*, the choice being non-deterministic and unfair in that an infinite number of calls may be made to *stutter*.

There are two component computations, *halt* and the recursive call on *stutter*, that are combined through non-deterministic choice. As we have shown in Section 3.3.2, the transformer corresponding to choice is set union. The spec of halt is $\{H[\,]\}_*$. Let $x$ stand for the spec of *stutter*. The recursive call preceded by event *tl* is $cons(tl, x)$; see Section 3.3.5 (page 11) for a definition of *cons*. Thus, we have:

$$x = \{H[\ ]\}_* \cup cons(tl, x)$$

Observe that each of the transformers, $\cup$ and *cons* are smooth. So, their composition given above is smooth.

The steps in the application of the least fixed-point theorem successively yield, $\{W[\,]\}$, $\{W[\,], H[\,], W[tl]\}$, $\{W[\,], H[\,], W[tl], H[tl], W[tl^2]\}$, $\cdots$, and for any $i$, $\{H[tl^j] \mid 0 \le j \le i\}_* \cup \{W[tl^{i+1}]\}$. Then $lfp(stutter)$, the lub of this sequence, is $\{H[tl^i] \mid 0 \le i\}_*$.

### 4.1.2 The need for upward-closure of specs

From $lfp(stutter)$ we may deduce that every execution of *stutter* is finite, though unbounded, in length. But this is not what happens in reality. It is possible for an unfair coin to land tails forever, so the trace $D[tl^\omega]$ ought to be included in the spec. And, $\{H[tl^i] \mid 0 \le i\}_*$ is actually the spec of *stutter* where a *fair* coin is used in the toss so that there is no infinite computation.

The present difficulty arises because subset ordering over specs implies that the lub of a chain of specs is simply their union. We overcome this difficulty by introducing the notion of upward-closed specs that include the limits of countable chains of traces in the spec. The lub of a chain of upward-closed specs is not simply their union, but the upward-closure of their union. Thus, the lub of the specs $\{H[tl^j] \mid 0 \le j \le i\}_* \cup \{W[tl^{i+1}]\}$, for all $i$, $0 \le i$, is $\{H[tl^i] \mid 0 \le i\}_* \cup \{D[tl^\omega]\}$.

This discussion suggests that in solving $x = f(x)$, the transformer $f$ needs to transform an upward-closed spec to an upward-closed spec. Not all smooth transformers have this property. So, we introduce *bismooth transformers*, a subclass of smooth transformers, that have this property. We develop the appropriate concepts of upward-closure, and revisit the least fixed-point theorem.

## 4.2 Upward-Closure

### 4.2.1 Definitions

The following definitions make use of chains and their limits from Section 2.4.2 (page 8).

**Upward-closure** The upward-closure of spec $p$ is:

$p^* = \{lim(c) \mid c \text{ a chain in } p\}$.

It follows that $c^*$, the *upward-closure* of chain $c$, is $c \cup lim(c)$. In particular, for finite $c$, $c^* = c$. A spec is *upward-closed* if $p^* = p$, i.e., if chain $c$ is in $p$, then so is $lim(c)$.

Trace $s$ in $p$ is *maximal* if there is no $t$ in $p$ such that $s < t$. An arbitrary spec may not have a maximal trace, for example the spec $\{W[a^i] \mid i \geq 0\}$. But $p^*$ always has a maximal trace. Limit of a spec is given by:

$lim(p) = \{s \mid s \text{ a maximal trace in } p^*\}$

**Chain Continuity** Transformer $f$ is *chain continuous* if $f(c^*) = f^*(c)$ for any chain $c$ ($f^*(c)$ is $(f(c))^*$). Each of the following conditions imply chain continuity: (1) $f(lim(c)) = lim(f(c))$, for any chain $c$, (2) $f(t) = lim(f(t_{*'}))$ for any infinite trace $t$.

### 4.2.2 Properties of upward-closure

1. Upward-closure is algebraic closure, i.e, for specs $p$ and $q$,
   (a) (extensive) $p \subseteq p^*$.
   From the definition.
   (b) (monotonic) $p \subseteq q \;\Rightarrow\; p^* \subseteq q^*$.
   From the definition.
   (c) (idempotent) $(p^*)^* = p^*$.
   Apply definition of upward-closure noting that $p$ and $p^*$ have the same set of chains.

2. Alternate characterizations of upward-closure: For spec $p$,
   (a) $p^* = p \cup \{lim(c) \mid c \text{ an infinite chain in } p\}$.
   The limit of every finite chain of $p$ is in $p$.
   (b) $p^* = p \cup lim(p)$.
   Proof in Proposition 11 (page 37) in the Appendix.
   (c) $p^* = lim_*(p)$.
   Proof in Proposition 12 (page 37) in the Appendix. It follows that given spec $p$, $p^*$ is a spec because $p^* = lim_*(p)$, and $lim_*(p)$ is prefix-closed.

3. (Galois Adjoints) Finite prefix closure and upward-closure are Galois Adjoints, i.e., for traceset $p$ and spec $q$:

   $$p_{*'} \subseteq q \;\equiv\; p \subseteq q^*$$

   The following identities are then easily derived for specs $p$ and $q$.

   (a) $p_{*'} = (p^*)_{*'}$
   (b) $(p_{*'})^* = p^*$
   (c) $p_{*'} \subseteq q_{*'} \;\equiv\; p^* \subseteq q^*$
   (d) $p_{*'} = q_{*'} \;\equiv\; p^* = q^*$

4. (Distribution over union and intersection)

   (a) (Union) Let $F$ be a family of upward-closed specs and $P = \cup_{p \in F}(p)$. Then $P^* = \cup_{p \in F}(p^*)$ iff every chain in $P$ belongs to some spec in $F$. For finite $F$, $P^* = \cup_{p \in F}(p^*)$.
   (b) (Intersection) Let $F$ be a family of specs and $P = \cap_{p \in F}(p)$. Then $P^* = \cap_{p \in F}(p^*)$.
   (c) For any spec $q$, $q^* = \cup\{c^* \mid c \text{ a chain in } q\}$.

   Proof: See Proposition 13 (page 37) in the Appendix for the proof of item (4a). The result is of interest only if the chain is infinite and $F$ is infinite, because any finite chain in $P$ belongs to some spec in $F$, and for finite $F$ the result holds unconditionally. To see that $P^* = \cup_{p \in F}(p^*)$ does not hold unconditionally for infinite families, let $p_i = W_*[a^i]$, for every natural number $i$. Each $p_i$ is a spec, and $p_i^* = W_*[a^i]$. Therefore, $(\cup_i(p_i^*)) = \{W[a^i] \mid 0 \le i\}$. But, $(\cup_i p_i)^* = \{W[a^i] \mid 0 \le i\}^* = \{W[a^i] \mid 0 \le i\} \cup D[a^\omega]$.

   See Proposition 14 (page 37) for the proof of item (4b). The proof of item (4c) follows from item (4a), where the family consists of all the chains of $q$; note that $q = \cup\{c \mid c \text{ a chain in } q\}$, so $P = q$.

5. (upward-closure of tuples)
   For specs $p$ and $q$, $(p \times q)^* = p^* \times q^*$.

6. Let $f$ be a *finitely* smooth transformer, and $f(t) = lim(f(t_{*'}))$ for every infinite trace $t$. Then, $f$ is smooth.

   Proof: See Proposition 15 (page 38) in the Appendix.

7. Let $f$ be chain continuous. If a finite trace $s$ is in $f(t)$, for some trace $t$, then $s \in f(t_{*'})$. Equivalently, $f_{*'}(p) = f_{*'}(p^*) = f_{*'}(p_{*'})$, for any spec $p$.

   Proof: See Proposition 16 (page 38) in the Appendix.

8. For spec $p$ and filter $g$, $g(p^*) \subseteq g^*(p)$.

   Proof: See Proposition 17 (page 39).

## 4.3 Bismooth transformer

A smooth transformer does not necessarily preserve upward-closure. To see this, consider transformer $f$ where $f(t) = t_{*'}$, for all traces $t$. It is easy to see that $f$ is smooth. For an infinite chain of traces $c$, $f(c) = c$, so $f^*(c) = c^*$ whereas $f(c^*) = f(c \cup \{lim(c)) = f(c) \cup f(lim(c)) = c$.

Call a transformer *bismooth* if it preserves both upward and downward-closures. That is, for bismooth $f$:

| | |
|---|---|
| (smooth; preserves downward-closure) | $f(p_*) = f_*(p)$, for any traceset $p$, and |
| (preserves upward-closure) | $f(p^*) = f^*(p)$, for any spec $p$. |

A *finitely* bismooth transformer is smooth and it preserves upward-closure over *finitary* specs.

### 4.3.1 Example of a Bismooth Transformer

Consider transformer *or* from Section 3.3.2 (page 11) where *or* maps a tuple of specs $(p, q)$ to $p \cup q$. We have shown in that section that *or* is smooth. To prove that *or* is bismooth show that $or^*(p, q) = or((p, q)^*)$.

$$
\begin{array}{ll}
 & or((p,q)^*) \\
= & \{\text{upward-closure of tuples; item (5) (page 23)}\} \\
 & or(p^* \times q^*) \\
= & \{\text{rewriting}\} \\
 & or\{(x,y) \mid x \in p^*, y \in q^*\} \\
= & \{\text{definition of } or\} \\
 & \{x \mid x \in p^*\} \cup \{y \mid y \in q^*\} \\
= & \{\text{set theory}\} \\
 & p^* \cup q^* \\
= & \{\text{upward-closure distributes over finite union, item (4) (page 23)}\} \\
 & (p \cup q)^* \\
= & \{\text{definition of } or\} \\
 & or^*(p,q)
\end{array}
$$

### 4.3.2 Chain continuity $\neq$ Bismoothness

From its definition every bismooth transformer, even a finitely bismooth transformer, is chain continuous. In analogy with the definition of smooth transformers based on traces it may seem that we can give a similar characterization of bismooth transformers based on chains, namely, that every smooth and chain continuous transformer is bismooth. The following counterexample is due to Ernie Cohen.

Consider transformer *hide* from Section 3.3.3 (page 11) that was shown to be smooth. Let *hidea* be its instance that removes every $a$ event from a trace. It is not hard to see that $hidea(c^*) = hidea^*(c)$ for any chain $c$. Yet *hidea* is not bismooth, as shown below.

Let spec $p$ be $\{W[a^i b^i] \mid i \geq 0\}_*$, where $a$ and $b$ are different symbols from the event alphabet. Now $hidea^*(p) \neq hidea(p^*)$:

$$\begin{array}{ll} p^* = \{W[a^i b^i] \mid i \geq 0\}_* \cup D[a^\omega] & hidea(p^*) = \{W[b^i] \mid i \geq 0\} \\ hidea(p) = \{W[b^i] \mid i \geq 0\} & hidea^*(p) = \{W[b^i] \mid i \geq 0\} \cup D[b^\omega] \end{array}$$

### 4.3.3 Properties of Bismooth Transformers

As the counterexample in the previous subsection shows, chain continuity is insufficient for bismoothness. Typically, proving that a transformer is bismooth is considerably more difficult than proving that it is smooth. The properties given below simplify such proofs.

**Properties of Bismooth Transformers**

1. The identity transformer, $id(p) = p$, is bismooth.

2. (Bismooth composition) Composition of bismooth transformers is bismooth.

   Proof: Let $f$ and $g$ be bismooth and $p$ any spec. Then $f$ and $g$ are both smooth and their composition is smooth. The proof that $(f \circ g)(p^*) = (f \circ g)^*(p)$ is analogous to the corresponding result for smooth transformers, replacing all occurrences of downward-closure by upward-closure.

3. A transformer is bismooth if and only if it finitely bismooth.

   Proof: See Proposition 18 (page 39) in the Appendix.

4. Smooth transformer $f$ is bismooth if and only if (1) $f$ is chain continuous, and (2) corresponding to any chain $d$ in $f(p)$, where $p$ is a spec, there is a chain $c$ in $p$ such that $d \subseteq f(c)$.

   Proof: The proof of the "if" part is in Proposition 19 (page 39) in the Appendix. The proof of "only if" is in Proposition 20 (page 40) in the Appendix.

5. (Sufficient condition for bismoothness) Define a transformer to be *co-finite* if it maps only a finite number of finite traces to any finite trace. A transformer that is smooth, co-finite and chain continuous is bismooth.

   Proof: See Proposition 22 (page 41) in the Appendix.

Property (1) is easy to prove. Property (2), bismooth composition, permits definition of new bismooth transformers using the existing ones. Property (3) simplifies many proofs regarding bismooth transformers by eliminating considerations of infinite traces in a spec. Even though chain continuity by itself is insufficient to guarantee bismoothness, Property (4) shows that an additional condition on chains in $p$ and $f(p)$ is both necessary and sufficient for bismoothness. Property (5), a sufficient condition for bismoothness, is immensely helpful in proofs when a transformer is defined without using any known bismooth transformer. Almost all proofs in Section 4.3.4 (page 26) about the elementary transformers use this sufficient condition. Its proof uses a variation of Koenig's lemma which is given in Proposition (21) (page 40). The co-finiteness condition in property (5) is not necessary for bismoothness; for example if $f(t) = \{W[\,]\}$ for all $t$ then $f$ is bismooth though not co-finite.

### 4.3.4 Bismoothness of Transformers from Section 3.3

We showed a number of useful transformers in Section 3.3 (page 10). All transformers of that section except *hide* of Section 3.3.3 (page 11), discontinuous *filter* of Section 3.3.6 (page 12) and fair merge of Section 3.3.12 (page 15) are bismooth; see Table 1 (page 26).

| Transformer | Bismooth? | Proof |
|---|---|---|
| status map | Yes | Proposition 23 (page 42) |
| choice | Yes | Section 4.3.1 (page 24) |
| hide | No | Section 4.3.2 (page 24) |
| drop | Yes | Proposition 24 (page 42) |
| cons | Yes | Proposition 25 (page 43) |
| discontinuous filter | No | Proposition 26 (page 43) |
| continuous filter | Yes | Proposition 27 (page 43) |
| restrict by inclusion | Yes | special case of continuous filter |
| restrict by exclusion | Yes | special case of continuous filter |
| restrict by precedence | Yes | special case of continuous filter |
| atom | Yes | special case of continuous filter |
| unfair merge | Yes | Proposition 28 (page 44) |
| fair merge | No | Proposition 29 (page 44) |
| replace | Yes | Proposition 30 (page 44) |
| rendezvous | Yes | composition of bismooth transformers |
| Sequential Composition | Yes | Proposition 31 (page 45) |

Table 1: Summary of Bismoothness of Elementary Transformers

## 4.4 Least Upward-Closed Fixed Points of Bismooth Transformers

As shown in Section 4.1 (page 20), the least fixed point of any smooth $f$, *lfp*$(f)$, under subset ordering over specs is $\cup_i(f^i(W[\,]))$. Now, apply the fixed point theorem taking subset ordering over *upward-closed* specs. The theorem can be applied only if transformer $f$ is bismooth, so that it creates a chain of upward-closed specs $f^i(W[\,])$, $i \geq 0$. The lub of this chain is *not* $\cup_i\{f^i(W[\,])\}$, but $(\cup_i\{f^i(W[\,])\})^*$.

Formally, $p$ is an *upward-closed fixed point* if $p$ is both a fixed-point and upward-closed. The least upward-closed fixed-point (*lufp*) of $f$, written as *lufp*$(f)$, is *lfp*$^*(f)$. Note that *lufp*$(f)$ is a spec.

**Theorem 2** [Least Upward-Closed Fixed Point Theorem]
For bismooth $f$, $\mathit{lufp}(f) = \mathit{lfp}^*(f)$

Proof: See Proposition 32 (page 46) in the Appendix, for a direct proof.

Note that $\mathit{lfp}(f)$ is a spec; therefore $\mathit{lufp}(f)$ is also a spec.

Consider the coin toss example of Section 4.1.1 (page 21) whose least fixed point is $\{H[tl^i] \mid 0 \leq i\}_*$. The least upward-closed fixed point corresponding to this fixed point is $\{H[tl^i] \mid 0 \leq i\}_* \cup \{D[tl^\omega]\}$, which faithfully describes the finite and infinite behaviors with an unfair coin.

## 4.5 Min-Max Fixed Points of Smooth Transformers

A smooth transformer that includes some aspect of fairness, say, a discontinuous filter, is not bismooth. We develop a theorem that gives a precise characterization of the appropriate least fixed points of smooth transformers.

A smooth transformer is monotonic; hence, using the Knaster-Tarski theorem [12], it has a least fixed point. However, this fixed point may not be upward-closed. Consider the coin toss example of Section 4.1.1 (page 21) that uses a *fair* coin so that an infinite run of tails is inadmissible. The recursive equation describing this component is

$$x = \mathit{fc}(\{H[\,]\}_* \cup \mathit{cons}(tl, x))$$

where transformer *fc* implements a fair coin and, hence, is a discontinuous filter. There is no upward-closed fixed point of this equation. The desired fixed-point is $\{H[tl^i] \mid 0 \leq i\}_*$, but it is not upward-closed. So, instead of upward-closed fixed point, we look for a least fixed-point that *includes as many limit traces as possible under the fairness constraint.*

For any smooth transformer $f$ define $p$ to be a *maximal* fixed point of $f$ if $p$ is the greatest fixed point of $f$ in $p^*$; i.e., $p$ includes as many traces as possible from $p^*$. Observe that the greatest fixed point in any traceset $q$ is the union of all fixed points in $q$, because union of fixed points is a fixed point for any tracewise transformer. The least maximal fixed point of $f$, $\mathit{mmfp}(f)$, also called the min-max fixed point, is: (1) a maximal fixed point of $f$, and (2) the least among all maximal fixed points of $f$. Theorem 3 shows that min-max fixed point exists for any smooth transformer.

The following equation $E(X)$, for a given $X$ and unknown $r$, is important in the study of min-max fixed point:

$$r = X \cap f(r). \qquad [E(X)]$$

**Theorem 3** [Min-Max Fixed Point Theorem] Let $f$ be a smooth transformer and $p = \mathit{lfp}(f)$. Then (1) $\mathit{mmfp}(f)$ is the greatest fixed point of $f$ in $p^*$. Further, (2) if $f(p^*) \subseteq p^*$, $\mathit{mmfp}(f)$ is the greatest solution of $E(p^*)$.

Proof of (1) is in Proposition 33 (page 46) and of (2) in Proposition 34 (page 47).

The condition $f(p^*) \subseteq p^*$ in (2) holds if $f$ is chain continuous, see Proposition 35 (page 47). We consider a class of "fair" transformers in the next section for which the condition in (2) holds, and we give stronger characterizations of min-max fixed points for such transformers.

This theorem shows that the min-max fixed point can be "computed" by first computing a least fixed point and then a greatest fixed point, but there is no need for nested fixed point computations. The computation of the least fixed point of $f$ is "semi-constructive" for all smooth transformers using the least fixed point theorem. Unfortunately, a smooth transformer is not necessarily continuous with respect to the greatest lower bound. So, the greatest solution of $E(p^*)$ can not be computed in the same manner. The greatest solution of $E(X)$, call it $t$, satisfies $t \subseteq f^i(X)$, for all $i \geq 0$, by induction on $i$. Therefore, $t \subseteq \cap_i(f^i(X))$. But, $\cap_i(f^i(X))$ itself may not be a fixed point.

The min-max fixed point theorem is a generalization of the least upward-closed fixed point theorem 2 (page 27). To see this, let $f$ be bismooth. Given $p = \mathit{lfp}(f)$, $f(p^*) = f^*(p) = p^*$. So, $p^*$ is a fixed point, therefore, the greatest fixed point in $p^*$. Hence, $\mathit{mmfp}(f) = p^*$, from the min-max fixed point theorem.

## 4.6 Fixed Point under Fairness

A common form of a smooth transformer is $g \circ h$ where $g$ is a discontinuous filter, typically modeling fairness, and $h$ a bismooth transformer. A stronger version of Theorem 3 (page 27) applies in this case.

**Theorem 4** [Min-Max Fixed Point Theorem under Fairness] Let $f = g \circ h$ where $g$ is a filter, $h$ is bismooth and $p = \mathit{lfp}(f)$. Then $\mathit{mmfp}(f)$ is the greatest solution of the equation $E(p^*)$, as well as of $E'(p^*)$, where $E'(X)$ is the equation $r = g(X) \cap h(r)$.

Proof: See Proposition 36 (page 47) in the Appendix. □

A special case of this theorem often arises in practice: for any infinite trace $t$, $t \in h(t)$. This holds for the coin-toss example shown previously in this section. In this case a simpler characterization exists for the min-max fixed point.

**Theorem 5** Let $f = g \circ h$ where $g$ is a filter, $h$ is bismooth, and for any infinite trace $t$, $t \in h(t)$. Then $\mathit{mmfp}(f) = g(p^*)$, where $p = \mathit{lfp}(f)$.

Proof: See Proposition 37 (page 48) in the Appendix.

# 5 Concluding Remarks

This paper grew out of an effort to develop a proof theory for Orc [4, 6, 13], a concurrent programming language designed by the author and his collaborators. The concepts developed during that work, such as smooth and bismooth transformers, were found to be applicable for concurrent systems in general. We have constructed the transformers for Orc constructs by combining some of the elementary transformers described here. We have also extended the theory to real time systems.

We are currently developing a proof theory for concurrent systems, based on the theory developed here. A spec is a predicate over traces. Each elementary transformer corresponds to some operation on one or more predicates; for example, choice is simply disjunction over predicates and a filter is a conjunction of the filter predicate to eliminate unacceptable traces. Other transformers, such as merge and rendezvous, and have no simple counterpart in predicate calculus though they can be specified using quantification.

**Related Work** Applying denotational semantics to a concurrency calculus was pioneered by Hoare and his collaborators for CSP [1]. In a series of papers, they have developed a number of models culminating in a failure-divergence model [2]. They have defined all the relevant features of CSP, including rendezvous-based synchronized communication as well as both internal and external non-determinism. Fairness is not modeled because it is not relevant for CSP.

The theory proposed in this paper is inherently asynchronous. Concurrent execution is modeled via interleaving of actions. Yet, it is possible to simulate rendezvous, as we show in Section 3.3.14 (page 17). There is no special treatment for failure in our theory because it can be included as part of the spec of a component.

The distinction between internal and external non-determinism is exemplified by the expressions $ab + ac$ and $a(b + c)$, where $a$ is an internal event of a component $X$, $b$ and $c$ are events on which $X$ synchronizes with another component $Y$, and $+$ denotes non-deterministic choice. In $ab + ac$ the choice is made internally by $X$ to synchronize on either the $b$ (if it has chosen the $ab$ alternative) or the $c$ event (with $ac$ alternative). If $X$ has chosen to synchronize on $b$ and $Y$ offers $c$, there is a deadlock. This distinction is modeled in our theory by $X$ executing an internal decision event, say a coin toss, that decides between $b$ and $c$ in $ab + ac$. The internal specification of $X$ includes the decision event as a visible event though it is invisible in the external spec. In $a(b + c)$, the choice of the synchronizing event is determined externally, by $Y$ offering either $b$ or $c$.

Broy and Nelson [3] includes a number of important results concerning the existence and non-existence of fixed-points in the presence of fair choice. Their paper develops the theory for the "dovetail" operator that combines fair choice with angelic non-determinism, so that a terminating computation causes competing non-terminating computations to be discarded and rolled-back.

Meseguer, in personal communication, has observed that the theory presented here is an instance of more general constructions in $\omega-$posets [14, 8, 9].

**Acknowledgment** I am truly grateful to José Meseguer whose thorough reading of an earlier draft, and substantive technical comments, especially about connections to category theory and $\omega$-cpo has given me a deeper understanding of my own work. Tony Hoare has been a constant source of encouragement and inspiration, particularly about the applicability of the work in concurrent program verification. I am grateful to Ernie Cohen who has spent considerable time working with me and helping with several conceptual issues. Manfred Broy pointed out some deficiencies in this theory, and pointed me to the relevant literature about fixed points under fairness. This work was inspired by an attempt to prove correctness of programs written in the Orc programming language [4, 6, 13]; I am grateful to the current and former Orc team members, in particular, William Cook, Adrian Quark, David Kitchin, John Thywissen and Arthur Peters. Vladimir Lifschitz has been a sounding board and advisor on many algebraic questions. Members of IFIP WG 2.3, as always, have given many helpful suggestions.

## A Appendix: Deatailed Proofs

**Proposition 1** A transformer $f$ is smooth if and only if it preserves prefix-closure over individual traces, i.e., $f_*(t) = f(t_*)$ for every trace $t$.

Proof: It is easy to see that if the given transformer is smooth then $f_*(t) = f(t_*)$ for every trace $t$, by replacing $p$ by $\{t\}$. In the other direction, given that $f_*(t) = f(t_*)$ for every trace $t$:

$$\begin{array}{ll}
 & f_*(p) \\
= & \{\text{definition of } f \text{ over a traceset}\} \\
 & (\cup\{f(t) \mid t \in p\})_* \\
= & \{\text{prefix-closure distributes over set union}\} \\
 & \cup\{f_*(t) \mid t \in p\} \\
= & \{\text{Assumption: } f(t_*) = f_*(t)\} \\
 & \cup\{f(t_*) \mid t \in p\} \\
= & \{\text{rewrite}\} \\
 & \cup\{\cup\{f(s) \mid s \in t_*\} \mid t \in p\} \\
= & \{\text{definition of } p_*\} \\
 & \cup\{f(s) \mid s \in p_*\} \\
= & \{\text{definition of } f \text{ over a set}\} \\
 & f(p_*)
\end{array}$$

**Proposition 2** A transformer is smooth if and only if it maps specs to specs.

Proof: A smooth transformer maps specs to specs, by definition. Next, we show that any transformer $f$ that maps specs to specs is smooth. We apply the induction principle from Section 2.2 (page 6) for this proof. Assume that for any trace $t$, $s < t$ implies $f(s_*) = f_*(s)$, then show that $f(t_*) = f_*(t)$. The proof consists of two parts by mutual inclusion.

Proof of $f(t_*) \subseteq f_*(t)$:

$$\begin{array}{ll}
 & f(t_*) \\
= & \{t_* = \{t\} \cup \{s_* \mid s < t\}\} \\
 & f(\{t\} \cup \{s_* \mid s < t\} \\
= & \{f \text{ is trace-wise}\} \\
 & f(t) \cup \{f(s_*) \mid s < t\} \\
= & \{\text{induction hypothesis}\} \\
 & f(t) \cup \{f_*(s) \mid s < t\} \\
\subseteq & \{\text{monotonicity of } f\text{: for } s < t,\ f_*(s) \subseteq f_*(t)\} \\
 & f(t) \cup f_*(t) \\
= & \{f(t) \subseteq f_*(t)\} \\
 & f_*(t)
\end{array}$$

Proof of $f_*(t) \subseteq f(t_*)$:

$$\begin{array}{ll} & f_*(t) \\ \subseteq & \{\{t\} \subseteq t_* \text{ and } f \text{ is trace-wise}\} \\ & f_*(t_*) \\ = & \{t_* \text{ is a spec and } f \text{ maps specs to specs. So, } f(t_*) \text{ is a spec.} \\ & \quad \text{Therefore, } f_*(t_*) = f(t_*)\} \\ & f(t_*) \end{array}$$

**Proposition 3** Composition of smooth transformers is smooth.

Proof: Let $f$ and $g$ be smooth transformers and $(f \circ g)$ their composition. For any traceset $p$ we show that $(f \circ g)(p_*) = (f \circ g)_*(p)$.

$$\begin{array}{ll} & (f \circ g)(p_*) \\ = & \{\text{definition of composition}\} \\ & f(g(p_*)) \\ = & \{g \text{ is smooth. So, } g(p_*) = g_*(p)\} \\ & f(g_*(p)) \\ = & \{f \text{ is smooth; apply } f \text{ to } g_*(p)\} \\ & f_*(g(p)) \\ = & \{f_*(g(p)) = (f(g(p)))_* = ((f \circ g)(p))_* = (f \circ g)_*(p)\} \\ & (f \circ g)_*(p) \end{array}$$

**Proposition 4** Any *statusmap* transformer of Section 3.3.1 (page 11) is smooth.

Proof: Let $f$ be a *statusmap* transformer and $y[m]$ be any trace.

$$\begin{array}{ll} & f_*(y[m]) \\ = & \{f(y[m]) = \{y'[m]\}\} \\ & \{y'[m]\}_* \\ = & \{\text{Closure expansion}\} \\ & \{y'[m]\} \cup W[m_{*'}] \end{array}$$

And also,

$$\begin{array}{ll} & f(y_*[m]) \\ = & \{\text{Closure expansion}\} \\ & f(\{y[m]\} \cup W[m_{*'}]) \\ = & \{f \text{ is trace-wise}\} \\ & \{f(y[m])\} \cup f(W[m_{*'}]) \\ = & \{f(y[m]) = \{y'[m]\},\ f(W[k]) = \{W[k]\}, \text{ for any } k\} \\ & \{y'[m]\} \cup W[m_{*'}] \end{array}$$

**Proposition 5** Transformer *or* of Section 3.3.2 (page 11) is smooth.

$$\begin{array}{ll}
 & s_* \ \mathit{or}\ t_* \\
= & \{\text{trace-wise transformer on two arguments}\} \\
 & \{u \ \mathit{or}\ v \mid u \in s_*,\ v \in t_*\} \\
= & \{\text{definition of } \mathit{or}\} \\
 & \cup\{\{u, v\} \mid u \in s_*,\ v \in t_*\} \\
= & \{\text{set theory}\} \\
 & s_* \cup t_* \\
= & \{\text{prefix-closure distributes over traceset union}\} \\
 & (\{s, t\})_* \\
= & \{s \ \mathit{or}\ t = \{s, t\}\} \\
 & (s \ \mathit{or}\ t)_*
\end{array}$$

**Proposition 6** Transformer *hide* of Section 3.3.3 (page 11) is smooth.

Proof: Let $f(t)$ denote $\mathit{hide}(E, t)$ for a specific $E$. The following fact is obvious:

$$x \leq f(t) \equiv (\exists s : s \leq t : f(s) = x) \qquad (1)$$

$$\begin{array}{ll}
 & x \in f_*(t) \\
\equiv & \{\text{definition of prefix-closure}\} \\
 & \{x \mid x \leq f(t)\} \\
\equiv & \{\text{from (1)}\} \\
 & (\exists s : s \leq t : f(s) = x) \\
\equiv & \{\text{definition of prefix-closure and } f \text{ applied to a traceset}\} \\
 & x \in f(t_*)
\end{array}$$

**Proposition 7** Transformer *cons* of Section 3.3.5 (page 11) is smooth.

Proof: Recall the definition of *cons*:

$$\begin{array}{l}
\mathit{cons}(a, W[\,]) = \{W[\,], W[a]\} \\
\mathit{cons}(a, y[m]) = \{y[am]\}
\end{array}$$

Let $f$ be an instance of *cons* for some event $a$. We prove $f_*(y[m]) = f(y_*[m])$.

$$\begin{array}{ll}
 & f(y_*[m]) \\
= & \{\text{Closure expansion}\} \\
 & f(\{y[m]\} \cup W[m_{*'}]) \\
= & \{f \text{ is trace-wise}\} \\
 & f(y[m]) \cup f(W[m_{*'}]) \\
= & \{\text{definition of } f\text{. The term } W[\,] \text{ below is from } f(W[\,])\} \\
 & \{y[am]\} \cup \{W[\,]\} \cup W[am_{*'}] \\
= & \{\text{rewriting}\} \\
 & \{y[am]\} \cup W[(am)_{*'}] \\
= & \{\text{Closure expansion}\} \\
 & y_*[am] \\
= & \{\text{definition of } f\text{. Equality holds for } m = [\,] \text{ as well.}\} \\
 & f_*(y[m])
\end{array}$$

**Proposition 8** Unfair merge transformer of Section 3.3.11 (page 14) is smooth.

Proof:

• $(y[m])_* \mid (z[n])_* = (y[m] \mid z[n])_*$, where $y$ and $z$ are from $\{H,\ W\}$:
Then $m$ and $n$ are both finite and, hence, $m_{*'} = m_*$ and $n_{*'} = n_*$.

$$
\begin{array}{cl}
 & (y[m])_* \mid (z[n])_* \\
= & \{\text{Closure expansion; replace } m_{*'} \text{ by } m_* \text{ and } n_{*'} \text{ by } n_*.\} \\
 & (\{y[m]\} \cup W[m_*]) \mid (\{z[n]\} \cup W[n_*]) \\
= & \{\mid \text{ is trace-wise, so distributes over set union}\} \\
 & \{y[m] \mid z[n]\} \cup \{y[m] \mid W[n_*]\} \cup \{W[m_*] \mid z[n]\} \cup \{W[m_*] \mid W[n_*]\} \\
= & \{\text{definition of } \mid\} \\
 & (y \cap z)(m \otimes n) \cup W[m \otimes n_*] \cup W[m_* \otimes n] \cup W[m_* \otimes n_*] \\
= & \{(m \otimes n_*) \subseteq (m_* \otimes n_*),\ (m_* \otimes n) \subseteq (m_* \otimes n_*)\} \\
 & (y \cap z)(m \otimes n) \cup W[m_* \otimes n_*] \\
= & \{\otimes \text{ distributes over prefixes; see Section 3.3.11, page 15}\} \\
 & (y \cap z)(m \otimes n) \cup W[(m \otimes n)_*] \\
= & \{\text{For finite } m \text{ and } n,\ (m \otimes n)_* = (m \otimes n)_{*'}\} \\
 & (y \cap z)(m \otimes n) \cup W[(m \otimes n)_{*'}] \\
= & \{\text{Closure expansion}\} \\
 & ((y \cap z)(m \otimes n))_* \\
= & \{\text{definition of } \mid\} \\
 & (y[m] \mid z[n])_*
\end{array}
$$

• $(D[m])_* \mid (z[n])_* = (D[m] \mid z[n])_*$, where $z$ is from $\{H,\ W\}$:

Then $n$ is finite and $n_{*'} = n_*$.

$$
\begin{array}{cl}
 & (D[m])_* \mid (z[n])_* \\
= & \{\text{Closure expansion; replace } n_{*'} \text{ by } n_*.\} \\
 & (\{D[m]\} \cup W[m_{*'}]) \mid (\{z[n]\} \cup W[n_*]) \\
= & \{\mid \text{ is trace-wise, so distributes over set union}\} \\
 & \{D[m] \mid z[n]\} \cup \{D[m] \mid W[n_*]\} \cup \{W[m_{*'}] \mid z[n]\} \cup \{W[m_{*'}] \mid W[n_*]\} \\
= & \{\text{definition of } \mid\} \\
 & D(m \otimes n_*) \cup D[m \otimes n_*] \cup W[m_{*'} \otimes n] \cup W[m_{*'} \otimes n_*] \\
= & \{m_{*'} \otimes n \subseteq m_{*'} \otimes n_*\} \\
 & D[m \otimes n_*] \cup W[m_{*'} \otimes n_*]
\end{array}
$$

And,

$$
\begin{array}{cl}
 & (D[m]) \mid z[n])_* \\
= & \{\text{definition of } \mid\} \\
 & D_*[m \otimes n_*] \\
= & \{\text{Closure expansion}\} \\
 & D[m \otimes n_*] \cup W[(m \otimes n_*)_{*'}] \\
= & \{(m \otimes n_*)_{*'} = (m_{*'} \otimes n_{*'}); \text{ replace } n_{*'} \text{ by } n_*\} \\
 & D[m \otimes n_*] \cup W[m_{*'} \otimes n_*]
\end{array}
$$

- $D_*[m] \mid D_*[n] = (D[m] \mid D[n])_*$:

$$\begin{array}{cl}
 & D_*[m] \mid D_*[n] \\
= & \{\text{Closure expansion}\} \\
 & (\{D[m]\} \cup \{W[m_{*'}]\}) \mid (\{D[n]\} \cup \{W[n_{*'}]\}) \\
= & \{\mid \text{ is trace-wise, so distributes over set union}\} \\
 & \{D[m] \mid D[n]\} \cup \{D[m] \mid W[n_{*'}]\} \cup \{W[m_{*'}] \mid D[n]\} \cup \{W[m_{*'}] \mid W[n_{*'}]\} \\
= & \{\text{definition of } \mid\} \\
 & D[m \otimes n_*] \cup D[m_* \otimes n] \cup D[m \otimes n_{*'}] \cup D[m_{*'} \otimes n] \cup W[m_{*'} \otimes n_{*'}] \\
= & \{m \otimes n_{*'} \subseteq m \otimes n_*,\ m_{*'} \otimes n \subseteq m_* \otimes n\} \\
 & D[m \otimes n_*] \cup D[m_* \otimes n] \cup W[m_{*'} \otimes n_{*'}]
\end{array}$$

And,

$$\begin{array}{cl}
 & (D[m] \mid D[n])_* \\
= & \{\text{definition of } \mid\} \\
 & (D[m \otimes n_*] \cup D[m_* \otimes n])_* \\
= & \{\text{prefix-closure distributes over set union}\} \\
 & D_*[m \otimes n_*] \cup D_*[m_* \otimes n] \\
= & \{\text{Closure expansion}\} \\
 & D[m \otimes n_*] \cup W[(m \otimes n_*)_{*'}] \cup D[m_* \otimes n] \cup W[(m_* \otimes n)_{*'}] \\
= & \{(m \otimes n_*)_{*'} = m_{*'} \otimes n_{*'},\ (m_* \otimes n)_{*'} = m_{*'} \otimes n_{*'}\} \\
 & D[m \otimes n_*] \cup D[m_* \otimes n] \cup W[m_{*'} \otimes n_{*'}]
\end{array}$$

**Proposition 9** Transformer *replace* of Section 3.3.13 (page 16) is smooth.

Proof: Recall the definition of *replace*. The transformer is denoted by $f$, $\sigma$ is a generic source and $\tau$ the corresponding target. Symbol $a$ is a generic event. The definition is given in clausal form in a functional style, where the clauses are attempted in the given order from top to bottom.

$$\begin{array}{lll}
f(y[\sigma']) & = & y[\,], \text{ where } \sigma' \text{ is a proper prefix of a source} \\
f(y[\sigma m]) & = & \{cons(\tau, f(y[m])) \mid (\sigma,\tau) \in R\} \\
f(y[am]) & = & cons(a, f(y[m])),\ a \notin \text{source alphabet}
\end{array}$$

First, we show that $f(t_*) = f_*(t)$ for any finite trace $t$. The proof uses induction on the length of $t$. Since the trace is finite, we use general prefix instead of finite prefix operator in the proofs. Consider the different possible forms of $t$.

1. $t = y[\sigma']$: $f(t) = y[\,]$

   $f(t_*) = f(y_*[\sigma']) = f\{y[\sigma'], W[\sigma'_*]\} = f(y[\sigma']) \cup f(W[\sigma'_*]) = \{y[\,], W[\,]\}$
   $f_*(t) = y_*[\,] = \{y[\,], W[\,]\}$

2. $t = y[\sigma m]$: $f(t) = \{cons(\tau, f(y[m])) \mid (\sigma,\tau) \in R\}$

$t_* = y_*[\sigma m] = y[\sigma m] \cup W[(\sigma m)_*] = y[\sigma m] \cup W[\sigma_* - \{\sigma\}] \cup W[\sigma m_*]$
$f(t_*) = \{cons(\tau, f(y[m])) \mid (\sigma,\tau) \in R\} \cup \{W[\,]\} \cup \{cons(\tau, f(W[m_*])) \mid (\sigma,\tau) \in R\}$, or
$f(t_*) = \{W[\,]\} \cup \{cons(\tau, f(\{y[m]\} \cup W[m_*])) \mid (\sigma,\tau) \in R\}$

And,

$$\begin{array}{cl}
 & f_*(t) \\
= & \{f(t) = \{cons(\tau, f(y[m])) \mid (\sigma,\tau) \in R\}\} \\
 & \{cons(\tau, f(y[m])) \mid (\sigma,\tau) \in R\}_* \\
= & \{\text{rewrite}\} \\
 & \{W[\,]\} \cup \{cons(\tau, f_*(y[m])) \mid (\sigma,\tau) \in R\} \\
= & \{\text{induction: } f_*(y[m]) = f(y_*[m])\} \\
 & \{W[\,]\} \cup \{cons(\tau, f(y_*[m])) \mid (\sigma,\tau) \in R\} \\
= & \{\text{closure expansion}\} \\
 & \{W[\,]\} \cup \{cons(\tau, f(\{y[m]\} \cup W[m_*])) \mid (\sigma,\tau) \in R\}
\end{array}$$

3. $t = y[am]$: For $a \notin$ source alphabet, replace $\tau$ by $a$ in the proof above.

The smoothness result for infinite $t$ follows from item (6) of section 4.2.2 (page 22).

**Proposition 10** Sequential composition (transformer ; ) of Section 3.3.15 (page 17) is smooth.

$$\begin{array}{cl}
 & (H[m] \;;\; z[n])_* \\
= & \{\text{definition of } ;\,\} \\
 & (z[mn])_* \\
= & \{\text{Closure expansion}\} \\
 & \{z[mn]\} \cup W[(mn)_{*'}] \\
= & \{(mn)_{*'} = m_* \cup (mn_{*'}); \text{ note that } m \text{ is finite}\} \\
 & \{z[mn]\} \cup W[m_* \cup (mn_{*'})] \\
= & \{\text{distribute } W \text{ over the two terms in its argument}\} \\
 & \{z[mn]\} \cup W[m_*] \cup W[mn_{*'}]
\end{array}$$

And,

$$\begin{array}{cl}
 & H[m]_* \;;\; z[n]_* \\
= & \{\text{Closure expansion; note that } m \text{ is finite}\} \\
 & (H[m] \cup W[m_*]) \;;\; (z[n] \cup W[n_{*'}]) \\
= & \{\text{apply } ; \text{ trace-wise}\} \\
 & \{H[m] \;;\; z[n]\} \cup (H[m] \;;\; W[n_{*'}]) \cup (W[m_*] \;;\; z[n]) \cup (W[m_*] \;;\; W[n_{*'}]) \\
= & \{\text{rewrite}\} \\
 & \{z[mn]\} \cup W[mn_{*'}] \cup W[m_*] \cup W[m_*] \\
= & \{\text{rewrite}\} \\
 & \{z[mn]\} \cup W[m_*] \cup W[mn_{*'}]
\end{array}$$

The remaining proof, that $(s \;;\; z[n])_* = (s_* \;;\; z[n]_*)$, is straightforward.

**Proposition 11** For spec $p$, $p^* = p \cup lim(p)$.

Proof: Every maximal trace of $p^*$ is in $lim(p)$, by definition of $lim(p)$, and every non-maximal trace of $p^*$ is finite, and hence in $p$. Therefore, $p^* \subseteq p \cup lim(p)$. Conversely, $p \subseteq p^*$, from item (2a) (page 22), and $lim(p) \subseteq p^*$ by definition of $lim(p)$. So, $p \cup lim(p) \subseteq p^*$.

**Proposition 12** For any spec $p$, $p^* = lim_*(p)$.

Proof:

$$\begin{array}{cl} & lim_*(p) \\ = & \{\text{definition of } lim(p)\} \\ & \{s \mid s \text{ a maximal trace in } p^*\}_* \\ = & \{\text{prefix-closure distributes over set union}\} \\ & \cup\{s_* \mid s \text{ a maximal trace in } p^*\} \\ = & \{\text{every trace in a set is a prefix of some maximal trace in that set}\} \\ & \{t \mid t \text{ a trace in } p^*\} \\ = & \{\text{set theory}\} \\ & p^* \end{array}$$

**Proposition 13** Let $F$ be a family of upward-closed specs and $P = \cup_{p \in F}(p)$. Then $P^* = \cup_{p \in F}(p^*)$ iff every chain in $P$ belongs to some spec in $F$. For finite $F$, $P^* = \cup_{p \in F}(p^*)$.

Proof: Suppose every chain in $P$ belongs to some spec in $F$.

$$\begin{array}{cl} & P^* \\ = & \{\text{definition of upward-closure}\} \\ & \{lim(c) \mid c \text{ a chain in } P\} \\ = & \{\text{every chain } c \text{ in } P \text{ belongs to some spec in } F. \\ & \quad \text{Conversely, every chain in any spec in } F \text{ is a chain in } P\} \\ & \cup_{p \in F}\{lim(c) \mid c \text{ a chain in } p\} \\ = & \{\text{definition of upward-closure}\} \\ & \cup_{p \in F}(p^*) \end{array}$$

Conversely, suppose there is a chain $c$ in $P$ that does not belong to any spec in $F$. Then $lim(c) \in P^*$ whereas for any $p$, since $c \not\subseteq p$, $lim(c) \notin p^*$. So, $P^* \neq \cup_{p \in F}(p^*)$.

For a finite family of specs, every infinite chain in $P$ has an infinite subset in some $p$, using the pigeon-hole principle. Since $p$ is a spec, if it includes an infinite subset of a chain, it includes the entire chain. The result then follows from the proof above.

**Proposition 14** Let $F$ be a family of specs and $P = \cap_{p \in F}(p)$. Then $P^* = \cap_{p \in F}(p^*)$.

Proof: For any trace $t$

$$\begin{array}{cl}
 & t \in P^* \\
\equiv & \{\text{using } \{t\} \text{ for } p \text{ and } P \text{ for } q \text{ in item (3) (page 23)}\} \\
 & t_{*'} \subseteq P \\
\equiv & \{P = \cap_{p \in F}(p)\} \\
 & t_{*'} \subseteq p, \text{ for every } p \text{ in } F \\
\equiv & \{\text{using } \{t\} \text{ for } p \text{ and } p \text{ for } q \text{ in item (3) (page 23)}\} \\
 & t \in p^*, \text{ for every } p \text{ in } F \\
\equiv & \{\text{set theory}\} \\
 & t \in \cap_{p \in F}(p^*)
\end{array}$$

**Proposition 15** Let $f$ be a *finitely* smooth transformer, and $f(t) = lim(f(t_{*'}))$ for every infinite trace $t$. Then, $f$ is smooth.

Proof: We show that for any infinite trace $t$, $f(t_*) = f_*(t)$.
Let $c = t_{*'}$. Then, $c_* = c$, and $f(c_*) = f(c)$. Since $f$ is smooth over finite traces $f(c_*) = f_*(c)$. Therefore, $f(c) = f_*(c)$, or $f(c)$ is a spec.

$$\begin{array}{cl}
 & f(t_*) \\
= & \{t_* = \{t\} \cup c;\ f \text{ is trace-wise}\} \\
 & f(t) \cup f(c) \\
= & \{\text{Assumption: } f(t) = lim(f(c))\} \\
 & lim(f(c)) \cup f(c) \\
= & \{\text{using } f(c) \text{ for } p \text{ in item (2b) of section 4.2.2 (page 22)}\} \\
 & f^*(c) \\
= & \{f(c) \text{ is a spec; item (2c) of section 4.2.2 (page 22): } f^*(c) = lim_*(f(c))\} \\
 & lim_*(f(c)) \\
= & \{\text{Assumption: } f(t) = lim(f(c))\} \\
 & f_*(t)
\end{array}$$

**Proposition 16** Let $f$ be chain continuous. If a finite trace $s$ is in $f(t)$, for some trace $t$, then $s \in f(t_{*'})$. Equivalently, $f_{*'}(p) = f_{*'}(p^*) = f_{*'}(p_{*'})$, for any spec $p$.

Proof: Let $c = t_{*'}$.

$$\begin{array}{cl}
 & s \in f(t) \\
\Rightarrow & \{c = t_{*'}, \text{ so } t \in c^*\} \\
 & s \in f(c^*) \\
\Rightarrow & \{f \text{ is chain continuous}\} \\
 & s \in f^*(c) \\
\Rightarrow & \{s \text{ finite}\} \\
 & s \in f(c) \\
\Rightarrow & \{c = t_{*'}\} \\
 & s \in f(t_{*'})
\end{array}$$

It follows that for any spec $p$, $s \in f_{*'}(p) \Rightarrow s \in f_{*'}(p_{*'})$; so, $f_{*'}(p) \subseteq f_{*'}(p_{*'})$. Conversely, since $p_{*'} \subseteq p$, $f_{*'}(p_{*'}) \subseteq f_{*'}(p)$; so $f_{*'}(p) = f_{*'}(p_{*'})$. Now, substitute $p^*$ for $p$ in this identity to get $f_{*'}(p^*) = f_{*'}((p^*)_{*'})$. Since $(p^*)_{*'} = p_{*'}$, from item (3a) (page 23), we get $f_{*'}(p) = f_{*'}(p_{*'}) = f_{*'}(p^*)$.

**Proposition 17** For spec $p$ and filter $g$, $g(p^*) \subseteq g^*(p)$.

Proof: Let $b$ be the filter predicate associated with $g$.

$$
\begin{array}{ll}
 & x \in g(p^*) \\
\Rightarrow & \{\text{definition of filter}\} \\
 & x \in p^* \wedge b(x) \\
\Rightarrow & \{\text{From item 3 (page 23), using } \{x\} \text{ for } p \text{ and } p \text{ for } q\} \\
 & x_{*'} \subseteq p \wedge b(x) \\
\Rightarrow & \{\text{From property [F2] of filter, Section 3.3.6 (page 12)}\} \\
 & x_{*'} \subseteq p \wedge b(x_{*'}) \\
\Rightarrow & \{\text{definition of filter}\} \\
 & x_{*'} \subseteq g(p) \\
\Rightarrow & \{\text{From item 3 (page 23), using } \{x\} \text{ for } p \text{ and } g(p) \text{ for } q\} \\
 & x \in g^*(p)
\end{array}
$$

**Proposition 18** A transformer is bismooth if and only if it is finitely bismooth.

Proof: Clearly a bismooth transformer is finitely bismooth. We prove the converse, that if $f$ is finitely bismooth then $f(p^*) = f^*(p)$ for any spec $p$ (that may contain infinite traces). Let $q$ be the set of finite traces in $p$, ie., $q = p_{*'}$.

$$
\begin{array}{ll}
 & f^*(p) \\
= & \{\text{using } f(p) \text{ for } p \text{ in item (3b) (page 23)}\} \\
 & (f_{*'}(p))^* \\
= & \{f \text{ is finitely bismooth, so chain continuous.} \\
 & \quad \text{Using item (7) (page 23)}\} \\
 & (f_{*'}(p_{*'}))^* \\
= & \{q = p_{*'}\} \\
 & (f_{*'}(q))^* \\
= & \{\text{using } f(q) \text{ for } p \text{ in item (3b) (page 23)}\} \\
 & f^*(q) \\
= & \{f \text{ is finitely bismooth}\} \\
 & f(q^*) \\
= & \{p_{*'} = q_{*'}.\ \text{Using item (3d) (page 23), } p^* = q^*\} \\
 & f(p^*)
\end{array}
$$

**Proposition 19** Suppose transformer $f$ is (S0) smooth, (S1) chain continuous, and (S2) corresponding to any chain $d$ in $f(p)$, where $p$ is any spec, there is a chain $c$ in $p$ such that $d \subseteq f(c)$. Then $f$ is bismooth.

$$
\begin{array}{ll}
 & f(p^*) \\
= & \{\text{from item (4c) (page 23)}\}
\end{array}
$$

$f(\cup\{c^* \mid c \text{ a chain in } p\})$
$=$ $\{f$ is trace-wise$\}$
$\cup\{f(c^*) \mid c \text{ a chain in } p\}$
$=$ $\{f$ is chain continuous$\}$
$\cup\{f^*(c) \mid c \text{ a chain in } p\}$
$=$ $\{p = \{c \mid c \text{ a chain in } p\}$. So, $f(p) = \cup_c(f(c))$.
Condition (S2): every chain $d$ in $\cup_c(f(c))$ is in some $f(c)$.
Use item (4a) (page 23)
$\}$
$(\cup\{f(c) \mid c \text{ a chain in } p\})^*$
$=$ $\{f$ is trace-wise. So, $f(p) = (\cup\{f(c) \mid c \text{ a chain in } p\})\}$
$f^*(p)$

**Proposition 20** Suppose transformer $f$ is bismooth. Then $f$ is (S0) smooth, (S1) chain continuous, and (S2) corresponding to any chain $d$ in $f(p)$, where $p$ is any spec, there is a chain $c$ in $p$ such that $d \subseteq f(c)$.

Proof: Both (S0) and (S1) hold from the definition of bismooth. Next, we show (S2). Let $p$ be any spec, $d$ a chain in $f(p)$ and $t = lim(d)$. Since $d \subseteq f(p)$, $t \in f^*(p) = f(p^*)$. Therefore, for some $s$ in $p^*$, we have $t \in f(s)$. We show that $s_{*'}$ is the desired chain $c$.

$t \in f(s)$
$\Rightarrow$ $\{$monotonicity of downward-closure$\}$
$t_* \subseteq f_*(s)$
$\Rightarrow$ $\{f$ is smooth$\}$
$t_* \subseteq f(s_*)$
$\Rightarrow$ $\{$from $t = lim(d)$, $d \subseteq t_*\}$
$d \subseteq f(s_*)$
$\Rightarrow$ $\{s_* = s_{*'} \cup \{s\}\}$
$d \subseteq f(s_{*'}) \cup f(s)$
$\Rightarrow$ $\{$for any $x$ in $d$, if $x \in f(s)$ then $x \in f(s_{*'})$, from item (7) (page 23)$\}$
$d \subseteq f(s_{*'})$

**Proposition 21** Let $S$ and $T$ be two non-empty trees with a bipartite relation, *cover*, from the nodes of $S$ to the nodes of $T$. For node $x$ in $S$ and $y$ in $T$, say that $x$ covers $y$ and $y$ is covered by $x$ whenever $(x, y) \in cover$. Suppose:
(C1) the set of nodes of $S$ that cover any node of $T$ is non-empty and finite, and
(C2) if $x$ covers $y$ then the ancestors of $x$ (which includes $x$) together cover the ancestors of $y$.
Then, every path in $T$ is covered by some path in $S$.

First, without loss in generality, add a new root $s$ to $S$, $t$ to $T$ and the pair $(s, t)$ to *cover*. Neither the hypotheses nor the conclusion are affected by this construction.

Let $p$ be a path in $T$ starting at $t$. Construct a tree $R$ from $S$ and $p$ as follows. The nodes of $R$ are $\{(x, y) \mid (x, y) \in \mathit{cover},\ y \in p\}$. Node $(x, y)$ is the parent of $(x', y')$, $y' \neq t$, where $y$ is the parent of $y'$ in $p$ and $x$ the ancestor of $x'$ closest to it that covers $y$. Such an $x$ exists because of condition (C2). Node $x$ may possibly be $x'$. Every node in $R$ except $(s, t)$ has a parent. Observe:

1. $R$ is a tree with root $(s, t)$. Every node of $p$ is the second component of a distinct node in $R$. Hence, if $p$ is infinite so is $R$.

2. Every node in $R$ has finite degree: node $(x, y)$ of $R$ has children of the form $(x', y')$ where $y'$ is the unique child of $y$ in $p$. From (C1), $y'$ is covered by a finite number of nodes.

3. Apply Koenig's lemma in conjunction with items (1) and (2) to establish the existence of an infinite path $q$ in $R$. Let $q_1$ and $q_2$ be the sequences of first and second components, respectively, of $q$. By construction, $q_2 = p$. And $q_1$ corresponds to a path of $S$ that covers $p$, because $(x, y)$ is the parent of $(x', y')$ in $q$ where $x$ is an ancestor of $x'$ in $S$. The path corresponding to $q_1$ is finite if some node of $S$ appears infinitely often in $q_1$.

**Proposition 22** Define a transformer to be *co-finite* if it maps only a finite number of finite traces to any finite trace. A transformer that is smooth, co-finite and chain continuous is bismooth.

Proof: We show that $f$ satisfies the sufficient conditions for bismoothness given in Proposition 19 (page 39) for any spec $p$. Conditions (S0) and (S1) are met by the hypotheses of this proposition. We next prove (S2), that for every chain $d$ in $f(p)$ there is a chain $c$ in $p$ such that $d \subseteq f(c)$. We use Proposition 21 (page 40) to establish this claim. Below, we show that the conditions (C1) and (C2) required by Proposition 21 are met.

Let $S$ and $T$ be the set of finite traces in specs $p$ and $f(p)$, respectively, in Proposition 21. Parent of any trace in either tree is its immediate predecessor in the prefix order. Thus, $x_*$ is the ancestors of trace $x$. And, the relation *cover* is $\{(x, y) \mid x \in p,\ y \in f(x)\}$. Since $f$ is chain continuous, every finite trace in $f(p)$ is mapped to by some finite trace in $p$, from item (7) (page 23). Therefore, every node in $T$ is covered by some node in $S$. Further, from co-finiteness of $f$, every trace in $f(p)$ is covered by a finite set of traces of $S$, thus satisfying condition (C1) of Proposition 21. We show that condition (C2), that if $x$ covers $y$ then ancestors of $x$ cover the ancestors of $y$, is met:

$$\begin{array}{ll}
 & x \text{ covers } y \\
= & \{\text{meaning of } \mathit{cover}\} \\
 & y \in f(x) \\
\Rightarrow & \{\text{prefix closure is monotonic}\} \\
 & y_* \subseteq f_*(x) \\
= & \{f \text{ is smooth}\} \\
 & y_* \subseteq f(x_*) \\
= & \{\text{meaning of } \mathit{cover}\}
\end{array}$$

$x_*$ covers $y_*$

= {meaning of ancestor}

ancestors of $x$ cover the ancestors of $y$

**Proposition 23** A status map transformer is bismooth.

Proof: We show that any status map transformer, $f$, satisfies the conditions in Proposition 22. Hence, it is bismooth.

1. A status map transformer is smooth, from Proposition 4 (page 32).
2. Clearly, $f$ is co-finite.
3. Chain continuity is seen easily for any finite chain. We show that for any infinite trace $t$ with associated chain $c$, $f(t) = lim(f(c))$. Trace $t$ being infinite is of the form $D[m]$ where $m$ is infinite; and, $f(D[m]) = D[m]$. Also, $f(c)$ is $c$ because every element of the chain has status $W$. So, $lim(f(c)) = lim(c) = D[m]$.

**Proposition 24** A drop transformer is bismooth.

Proof: First, we show that transformer *drop*$'$ that drops the first occurrence, if any, of a *specific* event $\tau$ in every trace is bismooth. To drop a finite set of events, compose the corresponding *drop*$'$ for each event individually. Since finite compositions of bismooth transformers is bismooth, *drop* is bismooth. Similarly, a transformer that drops the first $n$ occurrences of events from a finite event set is bismooth, by composing $n$ successive *drop*.

Henceforth, use $f(t)$ for *drop*$'(\tau, t)$; we show $f$ is bismooth. We appeal to Proposition 22 (page 41).

1. *drop*$'$ is a special case of *hide* and *hide* is smooth, from Proposition 6 (page 33).
2. To see that $f$ is co-finite, consider any finite trace $s$. Then $s \in f(t)$ if (1) $s$ has no $\tau$ event and $t$ is the same as $s$ with at most one $\tau$ event inserted somewhere within its event sequence, or (2) $s$ has a $\tau$ event and $t$ is the same as $s$ with a $\tau$ event inserted somewhere within its event sequence preceding the first $\tau$ event. In both cases, the number of possible traces $t$ is finite.
3. Chain continuity is seen easily for any finite chain. We show that $f(lim(c)) = lim(f(c))$ for any infinite chain $c$. Let $t = lim(c)$.

Case 1) $t$ does not include $\tau$: Then none of the traces in $c$ include $\tau$. We have $f(t) = t$, and $lim(f(c)) = lim(c) = t$.

Case 2) $t = D[a\tau m]$ for some finite sequence $a$ and infinite sequence $m$. Then $c = W[a_*] \cup W[a\tau m_{*'}]$.

$$
\begin{array}{ll}
 & lim(f(c)) \\
= & \{c = W[a_*] \cup W[a\tau m_{*'}]\} \\
 & lim(f(W[a_*] \cup W[a\tau m_{*'}])) \\
= & \{f \text{ trace-wise}\} \\
 & lim(f(W[a_*]) \cup f(W[a\tau m_{*'}])) \\
= & \{f(W[a_*]) = W[a_*],\ f(W[a\tau m_{*'}]) = W[am_{*'}]\} \\
 & lim(W[a_*] \cup W[am_{*'}]) \\
= & \{\text{definition of prefix-closure}\} \\
 & lim(W[(am)_{*'}]) \\
= & \{\text{definition of } lim\} \\
 & D[am] \\
= & \{t = D[a\tau m]\} \\
 & f(t)
\end{array}
$$

**Proposition 25** A cons transformer is bismooth.

Proof: We show that a cons transformer, $f$, satisfies the conditions in Proposition 22. Hence, it is bismooth.

1. *cons* is smooth, from Proposition 7 (page 33).
2. Clearly, $f$ is co-finite.
3. Chain continuity is seen easily for any finite chain. Let $c$ be an infinite chain in $p$ and $t = lim(c)$. Trace $t$, being infinite, is of the form $D[m]$. Every trace in $c$ is of the form $W[n]$ where $n \in m_{*'}$. So, $f(c) = f\{W[n] \mid n \in m_{*'}\} = \{W[]\} \cup \{W[an] \mid n \in m_{*'}\}$. Now, $f(D[m]) = D[am]$. And, $lim(f(c)) = D[am]$.

**Proposition 26** A discontinuous filter is not bismooth.

Proof: Consider a discontinuous filter $f$ that accepts all finite traces and rejects some infinite trace $t$. Let $c$ be the chain corresponding to $t$. Then for the chain $c$, $f(c^*) = c$ whereas $f^*(c) = c \cup lim(c)$, violating the bismoothness condition applied to spec $c$.

**Proposition 27** A continuous filter is bismooth.

Proof: We appeal to Proposition 19 (page 39).

(S0) Every filter is smooth.

(S1) Chain continuity follows from the definition of continuous filter.

(S2) We show that for any spec $p$ and chain $d$ in $f(p)$ there exists a chain $c$ in $p$ such that $d \subseteq f(c)$. For any filter $f$, $f(p) \subseteq p$. So, any chain $d$ in $f(p)$ is a chain in $p$ that fulfills the condition. Further, $d = f(d)$.

**Proposition 28** Unfair merge is bismooth.

Proof: We appeal to Proposition 22 (page 41). The definition of unfair merge from Section 3.3.11 (page 14) is:

$y[m] \mid z[n] = (y \cap z)(m \otimes n)$, where both $y$ and $z$ are from $\{H,\ W\}$
$D[m] \mid z[n] = D[m \otimes n_*]$, where $z$ is from $\{H,\ W\}$
$D[m] \mid D[n] = D[m \otimes n_*] \cup D[m_* \otimes n]$

1. Unfair merge is smooth, from Proposition 8 (page 34).

2. It is clear that $\mid$ is co-finite.

3. Chain continuity is seen easily for any finite chain. We show that for any infinite trace $t$ with associated chain $c$, $f(t) = lim(f(c))$. Chain $c$ consists of tuples from the Cartesian product of two specs. At least one of the component subchains in $c$ is infinite. So, $t = lim(c)$ is of the form $(D[m], W[n])$ for infinite $m$ or $(D[m], D[n])$ for infinite $m$ and $n$. The remaining case, $(W[m], D[n])$ for infinite $n$, is symmetric to $(D[m], W[n])$.

Case 1) $t = (D[m], W[n])$, where $m$ is infinite:

$f(t) = D[m] \mid W[n] = D[m \otimes n_*]$
$c = (D[m], W[n])_{*'} = \{(W[k], W[k']) \mid k \in m_{*'}, k' \in n_*\}$
$f(c) = \{W[k \otimes k'] \mid k \in m_{*'}, k' \in n_*\} = W[m_{*'} \otimes n_*]$
$lim(f(c)) = D[m \otimes n_*]$

Case 2) $t = (D[m], D[n])$, where $m$ and $n$ are infinite:

$f(t) = D[m] \mid D[n] = D[m \otimes n_*] \cup D[m_* \otimes n]$
$c = (D[m], D[n])_{*'} = \{(W[k], W[k']) \mid k \in m_{*'}, k' \in n_{*'}\}$
$f(c) = \{W[k \otimes k'] \mid k \in m_{*'}, k' \in n_{*'}\} = W[m_{*'} \otimes n_{*'}]$
$lim(f(c)) = D[m \otimes n_*] \cup D[m_* \otimes n]$

**Proposition 29** Fair merge is not bismooth.

Proof: Fair merge is a discontinuous filter applied to unfair merge. Since discontinuous filter is not bismooth, fair merge is not bismooth either.

**Proposition 30** *replace* is bismooth.

Proof: We appeal to Proposition 22 (page 41). Henceforth, let $f(t) = replace(R, t)$.

1. *replace* is smooth, from Proposition 9 (page 35).

2. To see that $f$ is co-finite, for any trace $s$, replace all target symbols by the corresponding sources in all possible ways. Since the number of target symbols is finite in a finite trace and the number of corresponding sources is finite, only a finite number of traces map to $s$ under $f$.

3. Chain continuity is seen easily.

**Proposition 31** Sequential composition is bismooth.

Proof: We repeat the definition of sequential composition from Section 3.3.15 (page 17)

$H[m]$ **;** $z[n] = z[mn]$,
$s$ **;** $z[n] = s$, otherwise

We appeal to Proposition 22 (page 41).

1. Sequential composition is smooth, from Proposition 10 (page 36).

2. Transformer **;** is co-finite because for any finite trace $z[m]$, $z[m] = H[m']$ **;** $z[n]$ where $m = m'n$, or $z[m] = W[m]$. In either case, only a finite number of traces map to $z[n]$.

3. Chain continuity is seen easily for any finite chain. An infinite chain $c$ for this 2-arity transformer is a pair of chains $(d, d')$. There are three cases to consider, and we show that for the corresponding transformer $f$ in each case $f(lim(c)) = lim(f(c))$, which implies chain continuity.

   (a) $d$ includes a trace of the form $H[m]$: Then $lim(d) = H[m]$, and $d'$ is an infinite chain with distinct elements. So, $lim(d') = D[n]$ for some $n$.

   $$\begin{array}{ll} lim(c) = (H[m], D[n]), & f(lim(c)) = H[m] \;\mathbf{;}\; D[n] = D[mn] \\ f(c) = W[(mn)_{*'}], & lim(f(c)) = D[mn] \end{array}$$

   (b) $d$ does not include a trace of the form $H[m]$ and $d$ is an infinite chain with distinct elements: Let $lim(d) = D[m]$, for some infinite $m$.

   $$\begin{array}{ll} lim(c) = (D[m], -), & f(lim(c)) = D[m] \\ f(c) = W[m_{*'}], & lim(f(c)) = D[m] \end{array}$$

   (c) $d$ does not include a trace of the form $H[m]$ and $d$ is a finite chain: Let $lim(d) = W[m]$, for some $m$. The proof is similar to that for the previous case with $W[m]$ in place of $D[m]$.

**Proposition 32** For bismooth $f$, $\mathit{lufp}(f) = \mathit{lfp}^*(f)$.

Proof: Let $p$ be an abbreviation for $\mathit{lfp}(f)$. We have to show that (1) $p^*$ is a fixed point of $f$, (2) $p^*$ is upward-closed, and (3) for any fixed point $q$ of $f$ that is upward-closed, $p^* \subseteq q$.

1. $f(p^*) = p^*$:

$$\begin{array}{cl} & f(p^*) \\ = & \{f \text{ is bismooth}\} \\ & f^*(p) \\ = & \{p \text{ is } \mathit{lfp}(f);\text{ so } f(p) = p\} \\ & p^* \end{array}$$

2. $p^*$ is upward-closed: $(p^*)^* = p^*$, from the idempotence of upward-closure.

3. $p^* \subseteq q$:

$$\begin{array}{cl} & p^* \\ \subseteq & \{p = \mathit{lfp}(f) \text{ and } q \text{ any fixed point of } f\text{, so } p \subseteq q. \\ & \quad \text{upward-closure is monotonic, so, } p^* \subseteq q^*\} \\ & q^* \\ = & \{q \text{ is upward-closed; so, } q^* = q\} \\ & q \end{array}$$

**Proposition 33** Let $f$ be a smooth transformer and $p = \mathit{lfp}(f)$. Then $\mathit{mmfp}(f)$ is the greatest fixed point of $f$ in $p^*$.

Proof: Since $f$ is smooth, $p$ is a spec and $p^*$ is defined. Further, $p^*$ includes at least one fixed point, namely $p$. Let $q$ be the greatest fixed point of $f$ in $p^*$; then it is the union of all fixed points of $f$ in $p^*$. We show that $q$ is the min-max fixed point.

First, $q$ is a fixed point of $f$ because union of fixed points is a fixed point. Second, $q$ is maximal because, $q \subseteq p^* \;\Rightarrow\; q^* \subseteq p^*$; since $q$ is the greatest fixed point of $f$ in $p^*$, it is the greatest fixed point of $f$ in $q^*$. Finally, $q$ is the least maximal fixed point because for any maximal fixed point $s$ of $f$, $q \subseteq s$:

$$\begin{array}{cl} & \mathit{true} \\ \Rightarrow & \{p = \mathit{lfp}(f) \text{ and } s \text{ is a fixed point of } f\} \\ & p \subseteq s \\ \Rightarrow & \{\text{apply upward-closure to both sides}\} \\ & p^* \subseteq s^* \\ \Rightarrow & \{q \subseteq p^*\} \\ & q \subseteq s^* \\ \Rightarrow & \{q \text{ is a fixed point of } f \text{ in } s^*;\ s \text{ is the greatest fixed point of } f \text{ in } s^*\} \\ & q \subseteq s \end{array}$$

**Proposition 34** Suppose $f(p^*) \subseteq p^*$. Then the greatest fixed point of $f$ in $p^*$ is the greatest solution of $E(p^*)$.

Proof: From Proposition 33 (page 46) the greatest fixed point of $f$ in $p^*$ exists.

Recall that $E(X)$ is the equation $r = X \cap f(r)$ in unknown $r$. We show that, given $f(X) \subseteq X$, $q$ is a fixed point of $f$ in $X$ iff it is a solution of $E(X)$. Therefore, the greatest fixed point of $f$ in $p^*$ is the greatest solution of $E(p^*)$.

$$
\begin{array}{ll}
 & q \text{ is a solution of } E(X) \\
\equiv & \{\text{definition of } E(X)\} \\
 & q = X \cap f(q) \\
\equiv & \{q = X \cap f(q) \text{ implies } q \subseteq X\} \\
 & q = X \cap f(q) \wedge q \subseteq X \\
\equiv & \{q \subseteq X \;\Rightarrow\; f(q) \subseteq f(X).\text{ From } f(X) \subseteq X,\ f(q) \subseteq X\} \\
 & q = f(q) \wedge q \subseteq X \\
\equiv & \{\text{simple deduction}\} \\
 & q \text{ is a fixed point of } f \text{ in } X
\end{array}
$$

**Proposition 35** Let $p$ be a fixed point of a chain continuous transformer $f$. Then $f(p^*) \subseteq p^*$.

Proof:

$$
\begin{array}{ll}
 & f(p^*) \\
= & \{\text{item (4c) (page 23)}\} \\
 & f(\cup\{c^* \mid c \text{ a chain in } p\}) \\
= & \{f \text{ is trace-wise. So, it distributes over union of tracesets}\} \\
 & \cup\{f(c^*) \mid c \text{ a chain in } p\} \\
= & \{f \text{ is chain continuous: } f(c^*) = f^*(c)\} \\
 & \cup\{f^*(c) \mid c \text{ a chain in } p\} \\
\subseteq & \{(c \subseteq p) \Rightarrow (f(c) \subseteq f(p)) \Rightarrow (f^*(c) \subseteq f^*(p))\} \\
 & \cup\{f^*(p) \mid c \text{ a chain in } p\} \\
\subseteq & \{\text{Each term in the set is } f^*(p).\text{ There is at least one term, } W[\,].\} \\
 & f^*(p) \\
= & \{p \text{ is a fixed point of } f\} \\
 & p^*
\end{array}
$$

**Proposition 36** Let $f = g \circ h$ where $g$ is a filter, $h$ is bismooth and $p = \mathit{lfp}(f)$. Then $\mathit{mmfp}(f)$ is the greatest solution of the equation (1) $E(p^*)$, as well as of (2) $E'(p^*)$, where $E'(X)$ is the equation $r = g(X) \cap h(r)$.

Proof: We first show that $f(p^*) \subseteq p^*$. Then (1) follows from the second part of Theorem 3 (page 27).

$$
\begin{array}{ll}
 & f(p^*) \\
= & \{f = g \circ h\} \\
 & g(h(p^*)) \\
= & \{h \text{ is bismooth}\}
\end{array}
$$

$$
\begin{array}{ll}
 & g(h^*(p)) \\
\subseteq & \{\text{From Proposition 17 (page 39), } g(h^*(p)) \subseteq g^*(h(p))\} \\
 & g^*(h(p)) \\
= & \{g^*(h(p)) = (g(h(p)))^* = f^*(p)\} \\
 & f^*(p) \\
= & \{p \text{ is a fixed point of } f\} \\
 & p^*
\end{array}
$$

To show (2), we prove that the rights sides of $E(X)$ and $E'(X)$ are identical.

$$
\begin{array}{ll}
 & X \cap f(r) \\
= & \{\text{using the definition of } f\} \\
 & X \cap g(h(r)) \\
= & \{\text{from property of filter, Section 3.3.6 (page 12)}\} \\
 & g(X \cap h(r)) \\
= & \{\text{from property of filter, Section 3.3.6 (page 12)}\} \\
 & g(X) \cap h(r)
\end{array}
$$

**Proposition 37** Let $f = g \circ h$ where $g$ is a filter, $h$ is bismooth, and for any infinite trace $t$, $t \in h(t)$. Then $mmfp(f) = g(p^*)$, where $p = lfp(f)$.

Proof: First, we show $g(p^*) \subseteq h(g(p^*))$. Let $i = p^* - p$; then $i$ is a set of infinite traces. Let $n$ be the subset of $i$ that $g$ accepts, so $g(i) = n$.

Since $p$ is a fixed point of $f$, $g(h(p)) = p$. Apply $g$ to both sides and note that $g$ is idempotent, so $g(h(p)) = g(p)$. Hence, $g(p) = g(h(p)) = p$. Further, since $g(h(p)) \subseteq h(p)$ and $g(h(p)) = p$, $p \subseteq h(p)$.

$$
\begin{array}{ll}
 & g(p^*) \\
= & \{p^* = p \cup i,\ g(p) = p,\ g(i) = n\} \\
 & p \cup n \\
\subseteq & \{\text{for infinite } t,\ t \in h(t).\ \text{So, } n \subseteq h(n)\} \\
 & p \cup h(n) \\
\subseteq & \{p \subseteq h(p).\ \text{So, } p \cup h(n) \subseteq h(p) \cup h(n).\ h \text{ is trace-wise}\} \\
 & h(p \cup n) \\
= & \{g(p^*) = p \cup n, \text{ from the first line of this proof}\} \\
 & h(g(p^*))
\end{array}
$$

As given by Theorem 4 (page 28), $mmfp(f)$ is the greatest fixed point of the equation $r = g(p^*) \cap h(r)$ in $r$. Any solution of this equation is a subset of $g(p^*)$. We show that $g(p^*)$ is a solution, therefore $mmfp(f) = g(p^*)$. Replace $r$ by $g(p^*)$ in the right side of the equation to get

$$
\begin{array}{ll}
 & g(p^*) \cap h(g(p^*)) \\
= & \{g(p^*) \subseteq h(g(p^*)), \text{ from the above proof}\} \\
 & g(p^*)
\end{array}
$$

which is the left side of the equation.

## References

[1] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, June 1984.

[2] S. D. Brookes, A. W. Roscoe, and D. J. Walker. An operational semantics for CSP. Technical report, Carnegie Mellon University.

[3] Manfred Broy and Greg Nelson. Adding fair choice to Dijkstra's calculus. *TOPLAS*, 16(3):924–938, May 1994.

[4] Jayadev Misra et. al. Orc language project. Web site. Browse at `http://orc.csres.utexas.edu`.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1984.

[6] David Kitchin, Adrian Quark, and Jayadev Misra. Quicksort: Combining concurrency, recursion, and mutable data structures. In A. W. Roscoe, Cliff B. Jones, and Ken Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing. Springer, 2010. Written in honor of Sir Tony Hoare's 75th birthday.

[7] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[8] J. Meseguer. *Completions, Factorizations and Colimits for Omega-posets*. Reports, U. of California, Los Angeles. 1978.

[9] J. Meseguer. Order completion monads. *Algebra Universalis*, 16(1):63–82, 1983.

[10] R. Milner. *Communication and Concurrency*. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall, 1989.

[11] D. Scott. Outline of a mathematical theory of computation. In *4th Annual Princeton Conference on Inform. Sc. and Systems*, pages 169–176, 1970.

[12] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[13] I. Wehrman, D. Kitchin, W. Cook, and J. Misra. A timed semantics of Orc. *Theoretical Computer Science*, 402(2-3):234–248, August 2008.

[14] J. Wright, E. Wagner, and J. Thatcher. A uniform approach to inductive posets and inductive closure. *Theoretical Computer Science*, 7:57 – 77, 1978.