Programming Research Group

LOGIC OF GLOBAL SYNCHRONY

Chen, Yifeng and J. W. Sanders

PRG-RR-01-01

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

Logic of Global Synchrony

Chen, Yifeng and J. W. Sanders

Programming Research Group, OUCL,
Parks Road, Oxford OX1 3QD, UK
{yc, jeff}@comlab.ox.ac.uk

January 23, 2001

Abstract

An intermediate-level specification notation is presented for use with BSP-style programming. It is achieved by extending pre-post semantics to reveal state at points of global synchronisation. That enables us to integrate the pre-post, finite and reactive-process styles of specification in BSP, as shown by our treatment of the dining philosophers. The language is provided with a complete set of laws and has been formulated to benefit from a simple predicative semantics.

1 Introduction

The approach of formal methods to sequential programming (i.e. the functional specification, correctness-preserving derivation and resulting valid implementation of systems) does not yet seem to have been applied with equal success to parallel programming. Semantic studies, dealing with the demonic nondeterminism resulting from abstraction, on which the sequential theory is based, are well developed for parallel programming. So the reason must lie elsewhere: perhaps in lack of uniform applicability — both across application domains and through levels of abstraction of the derivation process — of the various languages for parallel programming. It seems so far to have been premature to develop that 'formal methodology' when no language has gained the acceptance for parallel programming that Dijkstra's guarded-command language has gained for sequential programming.

The purpose of the present paper (with its companion [7] and the thesis [4]) is the investigation of such a formal method. The implementation language we choose is BSP [24, 23, 12]. Because BSP is targetted very much towards implementation at the expense of specification and reasoning, it fails to support abstraction in the way the guarded-command language does for sequential programming; see figure 1. As one particular instance we note that the parallel composition of BSP is neither a conjunction nor a disjunction, but a mixture of the two in the sense that it conjoins the behaviours of two processes but also generates non-determinism if the two processes try concurrently to write the same variable.

Thus we have extended BSP, following the style of [15], to a *hierarchy* of abstract languages which supports the incremental derivation of code from abstract functional specification. In this paper (and [7]) we present an intermediate-level language for describing and reasoning about BSP algorithms abstracted from the global puts and gets which BSP uses to implement global synchrony (for other languages see [4]). Since it has been designed to emphasise description (rather than execution, which is the purpose of BSP itself) we call the intermediate language a logic: the *logic of global synchrony*, LOGS.

The guarded-command language, like any sequential language, is supported by semantics (either relational or predicate-transformer, for example) in which a program is described by the way it transforms initial state to final state. In LOGS, in order to encompass reactive programs, we reveal the sequence of intermediate states of a computation at the points of global

specification	BSP program
potentially infinite logical formula	finite tree of executable commands
compositionality	no modularity
no explicit communication	explicit communication commands
intermediate states	synchronisation commands
logical compositions	more advanced compositions
unbounded non-determinism	finite non-determinism due to communication interference

Figure 1: Compromises in abstraction suffered by BSP

synchrony. That provides the abstraction necessary to describe the effect of BSP code but frees the software engineer from preoccupation with how such synchrony is achieved. It is for that reason that Logs is an *intermediate-level* notation. That design decision provides the basic Logs command, a computation with n points of global synchrony. If the computation terminates then the sequence of intermediates states is finite; if it is a reactive nonterminating computation then the sequence is infinite with no final state.

A complicated specification language makes formal reasoning unnecessarily complicated. Here the concern must be that the interposition of intermediate states in a computation has that effect at least on the semantics of Logs. An important feature of our work has been to provide a relatively simple predicative semantics and manner of demonstrating soundness with respect to it of the laws of the notation. That is achieved by the technique of *inheritance* which stems from [15]. Expressivity of the notation is demonstrated by the facility with which it supports the usual ways of reasoning about a reactive (not necessarily terminating) computation in terms of safety and liveness.

Whilst providing the new model for BSP we have taken the opportunity to rectify several deficiencies of existing models. The biggest extension is to allow task parallelism and MIMD programming; and also to allow unbounded nondeterminism. Further extensions include explicit modelling of local variables to supersede the dubious law $\mathbb{I} \parallel P = P$ [10], and replacement of lambda terms by an environmental alphabet to model better the sharing of global variables. The formal methods VDM and Z deal with pre-post specifications [20, 1] and have influenced our treatment, as has work on finite (i.e. terminating pre-post) BSP programs [10, 22, 18]. We have taken the view that it is critical for a real programming paradigm to deal with both data parallelism and task parallelism.

In section 2 LOGS is introduced, its laws given and their completeness verified. Important derived operators and their laws are set out in section 3. In section 4 we show LOGS, and in particular its derived operators for liveness and safety, at work on the benchmark example of the dining philosophers. The semantics is provided in section 5 where the technique of inheritance, for demonstrating soundness of the laws, is explained.

2 Specification language LOGS

2.1 Basic commands and combinators

No matter how useful it is, global synchronisation is expensive to implement on parallel/distributed systems. There is a case to be made for providing the BSP programmer with a language intermediate in abstraction between the standard low-level language (with explicit global synchronisations via the put and get events with respect to which they are implemented) and a more abstract one which abstracts them completely. For a language of the latter type, though important for requirements capture, does not enable a programmer to make tradeoffs based on efficiency.

In this paper we introduce language LOGS. It makes explicit the intermediate global states at synchronisation points, in such a way that the programmer is aware of the presence of global synchronisations without having to implement them (at this level). It also enables the programmer to reason, in a simple way, about safety and liveness properties.

On the other hand communications are abstracted in LOGS. That is to encourage the programmer to focus on the proposed effect of communications on the global state rather than to decide prematurely which communication commands to use. LOGS is thus an intermediate specification language, chosen for convenience, with no pretense at being as abstract as possible.

The basic commands of LOGS are listed in figure 2. There w denotes the list of all program variables; $\overleftarrow{w}$, pronounced 'pre-w', is the initial state (including input); $\overrightarrow{w}$, pronounced 'post-w', is the final state (including output); each w_k, $k < n$, denotes the state at the kth intermediate synchronisation

$\langle\, q \,\rangle_n$	nLOGS
$P \sqcap Q$	non-deterministic choice
$P\,;\,Q$	sequential composition
$P \sqcup Q$	parallel composition
$\mathtt{var}\ v\ P$	variable block
$\sim P$	negation

Figure 2: The basic commands of language LOGS

point, or *step*, with w_0 recording the first step after the initial state; and $n : \mathbb{N} \cup \{\infty\}$.

Predicate

$$q = q(\overleftarrow{w}, w_0, \ldots, w_{n-1}, \overrightarrow{w})$$

is a first-order predicate called the *internal predicate* of the n-step LOGS command $\langle\, q \,\rangle_n$. Such commands form the basic building blocks of LOGS since they specify reactive commands having n globally synchronous steps. The primitive command $\langle\, q \,\rangle_n$ starts in its initial state and, after n intermediate steps, terminates in its final state. If $n = \infty$ then the command becomes $\langle\, q(\overleftarrow{w}, w_0, w_1 \ldots, w_n, \ldots) \,\rangle_\infty$ without a final state. The set of all n-step commands is written nLOGS.

Nondeterminism and its dual parallel composition are standard, as is sequential composition and declaration of local blocks. A computation satisfies specification the negation $\sim P$ iff it does not satisfy specification P.

A LOGS command may consist of infinitely many LOGS commands or compositions. But each LOGS command is continuous.

The space of all predicates of a given signature forms a complete Boolean algebra whose conjunction, disjunction, negation and implication we write respectively as $\wedge$, $\vee$, $\neg$ and $\Rightarrow$. The least, or strongest, predicate we write `false` and the weakest we write `true`. For predicates p and q we write $p \leq q$ to mean that $p \Rightarrow q$ is a theorem.

2.2 Laws of LOGS commands

Command Q is a refinement of command P, $P \sqsubseteq Q$, iff $P \sqcap Q = P$. Following the definition of equivalence of predicates we define $P = Q$ iff $P \sqsubseteq Q$ and

$P \sqsupseteq Q$. LOGS commands form a complete Boolean algebra under $\sqcap$, $\sqcup$ and $\sim$. We write the top command as $\top$ and the bottom as $\perp$.

In this section we provide laws for LOGS which are sound with respect to the model to be introduced in section 5 (verification is either routine, in which case it is omitted, or is contained in [4]). In the present section we establish completeness of the laws.

Nondeterminism

Non-deterministic choice $\sqcap$ is idempotent, commutative and associative, so that the nondeterministic choice $\sqcap \mathcal{P}$ of a nonempty finite set $\mathcal{P}$ of commands is well defined. An empty nondeterministic choice is $\top$ so that in fact the nondeterministic choice of an *arbitrary* set $\mathcal{P}$ is also well defined.

Law 1
$$P \sqcap P \;=\; P \qquad (1)$$
$$P \sqcap Q \;=\; Q \sqcap P \qquad (2)$$
$$(P \sqcap Q) \sqcap R = P \sqcap (Q \sqcap R) \qquad (3)$$
$$\bigsqcap\{\,\} = \top \qquad (4)$$

The nondeterministic choice between two nLOGS commands is the disjunction of their internal predicates. That implies monotonicity of the embedding which takes a predicate to an nLOGS command.

Law 2
$$\langle\, p \,\rangle_n \sqcap \langle\, q \,\rangle_n = \langle\, p \vee q \,\rangle_n \qquad (1)$$
$$\langle\, p \,\rangle_n \sqsubseteq \langle\, q \,\rangle_n \quad \text{iff} \quad p \leq q \qquad (2)$$

In particular the weakest and strongest internal predicates give:

Law 3
$$\perp = \bigsqcap_{n \leqslant \infty} \langle\, \mathtt{true} \,\rangle_n \qquad (1)$$
$$\top = \langle\, \mathtt{false} \,\rangle_n \qquad (2)$$

Sequential composition

Sequential composition is associative.

Law 4 $(P\,;\,Q)\,;\,R \;=\; P\,;\,(Q\,;\,R)$

The sequential composition of an nLOGS command with an mGS command forms an $(n{+}m)$LOGS command.

Law 5
$$\langle\, p \,\rangle_n \,;\, \langle\, q \,\rangle_m = \left\langle \begin{array}{l} \exists \overline{w} \cdot p\,[\overline{w}/\overrightarrow{w}] \wedge \\ \quad q\,[\overline{w}, w_n, \ldots, w_{n+m-1}/\overleftarrow{w}, w_0, \ldots, w_{m-1}] \end{array} \right\rangle_{n+m}$$

Sequential composition distributes (arbitrary) non-determinism from both sides.

Law 6 $(\bigsqcap \mathcal{P}) \,;\, R = \bigsqcap\{P \,;\, R \mid P \in \mathcal{P}\}$ (1)

$R \,;\, (\bigsqcap \mathcal{P}) = \bigsqcap\{R \,;\, P \mid P \in \mathcal{P}\}$ (2)

In particular we infer from law **1**(4) that $\top \,;\, P = \top$, for any P.

Local blocks

Two consecutive local blocks can be merged.

Law 7 $\mathtt{var}\ v\ \mathtt{var}\ u\ P = \mathtt{var}\ v, u\ P$

A local block with a fresh variable achieves nothing.

Law 8 $\mathtt{var}\ v\ P = P$ if v is a fresh variable

A local block around an nLogs command becomes variable hiding of its internal predicate.

Law 9 $\mathtt{var}\ v\ \langle\, q \,\rangle_n \;=\; \langle\, \exists \overleftarrow{v}, v_0, \ldots, v_{n-1}, \overrightarrow{v} \cdot q \,\rangle_n$

Local block distributes (arbitrary) non-determinism.

Law 10 $\mathtt{var}\ v\ (\bigsqcap \mathcal{P}) = \bigsqcap\{\mathtt{var}\ v\ P \mid P \in \mathcal{P}\}$

In particular we infer from law **1**(4) that $\mathtt{var}\ v\ \top = \top$.

Parallel composition

Parallel composition $\sqcup$ is idempotent, commutative and associative so that the parallel composition $\sqcup \mathcal{P}$ of a nonempty finite set $\mathcal{P}$ of commands is well defined. An empty parallel composition is the bottom command $\perp$ so that in fact the parallel composition of an *arbitrary* set $\mathcal{P}$ of commands is well defined.

Law 11 $P \sqcup P \;=\; P$ (1)

$P \sqcup Q \;=\; Q \sqcup P$ (2)

$(P \sqcup Q) \sqcup R \;=\; P \sqcup (Q \sqcup R)$ (3)

$\bigsqcup\{\,\} = \perp$ (4)

In particular from law **6**(1) we infer that $\bot\,;\, P = \bot$, for any P. Thus $\bot$ and $\top$ are both left zeroes for sequential composition.

The parallel composition of two nLogs commands is the conjunction of their internal predicates, while that between two nLogs commands of different lengths is the top command.

Law 12 $\langle\, p \,\rangle_n \sqcup \langle\, q \,\rangle_n = \langle\, p \wedge q \,\rangle_n$ (1)
$\langle\, p \,\rangle_n \sqcup \langle\, q \,\rangle_m = \top$ (2)

In spite of law **2**(2), there is no simple equivalent of law **12**(2) for nondeterminism (as can be inferred from laws **15** and **16** below).

Non-determinism and parallel composition distribute each other.

Law 13 $(\bigsqcap \mathcal{P}) \sqcup R = \bigsqcap\{P \sqcup R \mid P \in \mathcal{P}\}$ (1)
$(\bigsqcup \mathcal{P}) \sqcap R = \bigsqcup\{P \sqcap R \mid P \in \mathcal{P}\}$ (2)

Negation

Because it is very much a *specification* construct, negation may be thought to be of marginal importance. However it is of great use to be able to specify a property as the negation of another. An important example is provided by its use in capturing negation (in the form $\neg P \vee Q$) in a rely-guarantee specification, which plays a critical role in compositional specifications.

Negation is idempotent.

Law 14 $\sim\sim P \;=\; P$

The following law calculates the negation of an nLogs command. In it i and n can be infinite.

Law 15 $\sim\langle\, p \,\rangle_n \;=\; \langle\, \neg p \,\rangle_n \sqcap \bigsqcap_{i \neq n} \langle\, \mathtt{true} \,\rangle_i$

De Morgan's laws hold for negation of (arbitrary) disjuction and conjunction so that either might have been introduced as a derived construct.

Law 16 $\sim\bigsqcap \mathcal{P} \;=\; \bigsqcup\{\sim P \mid P \in \mathcal{P}\}$ (1)
$\sim\bigsqcup \mathcal{P} \;=\; \bigsqcap\{\sim P \mid P \in \mathcal{P}\}$ (2)

2.3 Normal form and completeness

Commands in LOGS have a normal form:

Norm 1 $\quad \prod_{n \leqslant \infty} \langle\, q_n \,\rangle_n$

Lemma 1 *Any* LOGS *command may be reduced to normal form using (just) the laws of* LOGS.

Proof It suffices to demonstrate that nondeterminism can be moved to the front of any command, leaving an nLOGS term. Consider the different commands:

1. any nLOGS command is already in normal form

2. non-deterministic choice:

   $\prod_{n \leqslant \infty} \langle\, p_n \,\rangle_n \sqcap \prod_{n \leqslant \infty} \langle\, q_n \,\rangle_n \;=\; \prod_{n \leqslant \infty} \langle\, p_n \vee q_n \,\rangle_n$

3. sequential composition:

   $\prod_{i \leqslant \infty} \langle\, p_i \,\rangle_i \,;\, \prod_{j \leqslant \infty} \langle\, q_j \,\rangle_j \;=\; \prod_{n \leqslant \infty} \big\langle\, \bigvee_{i \leqslant n} (p_i \;;\; q_{n-i}) \,\big\rangle_n$

4. parallel composition:

   $\prod_{n \leqslant \infty} \langle\, p_n \,\rangle_n \sqcup \prod_{n \leqslant \infty} \langle\, q_n \,\rangle_n \;=\; \prod_{n \leqslant \infty} \langle\, p_n \wedge q_n \,\rangle_n$

5. variable block:

   $\mathtt{var}\ r\ \prod_{n \leqslant \infty} \langle\, p_n \,\rangle_n \;=\; \prod_{n \leqslant \infty} \langle\, \exists \overleftarrow{r}, r_0, \ldots, r_{n-1}, \overrightarrow{r} \cdot p_n \,\rangle_n$

6. negation:

   $$\begin{aligned} \sim \prod_{n \leqslant \infty} \langle\, p_n \,\rangle_n \;&=\; \bigsqcup_{n \leqslant \infty} \Big(\langle\, \neg p_n \,\rangle_n \sqcap \prod_{i \neq n} \langle\, \mathtt{true} \,\rangle_i \Big) \\ &=\; \prod_{n \leqslant \infty} \langle\, \neg p_n \,\rangle_n \end{aligned}$$

and the proof is complete. □

A LOGS command is said to be *finite* if it contains only a finite number of commands and compositions and if any nLOGS command it contains is finite.

Lemma 2 *For any finite* LOGS *command* P, *there exists* $N:\mathbb{N}\cup\{\infty\}$ *and predicates* $p_0, p_1, \ldots, p_N$, *such that either*

$$P \;=\; \bigsqcap_{n\leqslant N} \langle\, p_n \,\rangle_n \sqcap \bigsqcap_{n>N} \langle\, \texttt{false} \,\rangle_n$$

or

$$P \;=\; \bigsqcap_{n\leqslant N} \langle\, p_n \,\rangle_n \sqcap \bigsqcap_{n>N} \langle\, \texttt{true} \,\rangle_n \;\; .$$

Furthermore the reduction to that form may be achieved in finitely many steps using (just) the laws of LOGS.

The former command is called a *terminating finite norm*, while the latter is called a *non-terminating finite norm*. Negation converts one finite norm into the other. The proof of the lemma is similar to that of the previous lemma and so is omitted.

Theorem 1 (Completeness) *The laws of* LOGS *are complete for finite* LOGS *commands: semantic equality between two finite* LOGS *commands is provable using (just) the laws of* LOGS *and those of first-order logic.*

Proof Let P and Q be finite LOGS commands. According to Lemma 2, they may be reduced to either a terminating finite norm or a non-terminating finite norm.

If both P and Q are non-terminating finite norms (assume $N \leqslant M$),

$$\begin{aligned} P &= \textstyle\bigsqcap_{n\leqslant N} \langle\, p_n \,\rangle_n \sqcap \bigsqcap_{n>N} \langle\, \texttt{true} \,\rangle_n \\ Q &= \textstyle\bigsqcap_{m\leqslant M} \langle\, q_m \,\rangle_m \sqcap \bigsqcap_{m>M} \langle\, \texttt{true} \,\rangle_m \;\; , \end{aligned}$$

then they are semantically equal if and only if for any $k \leqslant N$,

$$\vdash \;\; (\forall \overleftarrow{w}, w_0, \ldots, w_{k-1}, \overrightarrow{w} \cdot p_k \Leftrightarrow q_k\,)$$

and for any k such that $N < k \leqslant M$,

$$\vdash \;\; (\forall \overleftarrow{w}, w_0, \ldots, w_{k-1}, \overrightarrow{w} \cdot q_k\,) \;\; .$$

Similarly P and Q can be compared if they are both terminating finite norms.

If P is terminating while Q is not, then their inequality can be immediately proved by checking that

$$P \sqcup \langle\, \mathtt{true} \,\rangle_\infty = \langle\, \mathtt{false} \,\rangle_n \ ,$$

whilst

$$Q \sqcup \langle\, \mathtt{true} \,\rangle_\infty = \langle\, \mathtt{true} \,\rangle_\infty \ .$$

□

Note that the set of laws provided here is not complete even if only one of the commands is infinite. In that case equality must be checked 'manually'.

We observe that the space of Logs commands is isomorphic to the space of predicates (perhaps infinite) on a sequence whose length is at least 2 (since the initial and the final states are always part of the sequence).

3 Derived Logs commands

The derived commands studied in this section are widely applicable and so support modularised specification and derivation in Logs.

We have already met the weakest $\bot$ and the strongest $\top$ in the previous section. They are called *chaos* and *magic* respectively.

0Logs commands and 1Logs commands are the special cases of nLogs for the cases n respectively 0 and 1. A 0Logs command contains no synchronisation, while a 1Logs command contains exactly one synchronisation point. Skip $\mathbb{I}$ is a 0Logs command which leaves the final state equal to the initial state. The three conditional commands are defined as usual in the refinement calculus.

Definition 1

$$\begin{array}{rcl} [\, q \,] & \hat{=} & \langle\, q \,\rangle_1 \\ \mathbb{I} & \hat{=} & \langle\, \overleftarrow{w} = \overrightarrow{w} \,\rangle_0 \\ (b)_\top & \hat{=} & \langle\, b(\overleftarrow{w}) \,\rangle_0 \sqcup \mathbb{I} \\ P \lhd b \rhd Q & \hat{=} & ((b)_\top \,;\, P) \sqcap ((\neg b)_\top \,;\, Q) \\ (b)_\bot & \hat{=} & \mathbb{I} \lhd b \rhd \bot \end{array}$$

One of the features of the extension from terminating to reactive programs is the part played by commands for pure termination and pure non-termination. $\rtimes$ terminates but after arbitrary behaviour; it is thus the weakest terminating computation.

$\bot$	chaos
$\top$	magic
$[\, q\,]$	1Logs
$\mathbb{I}$	skip
$(b)_\bot$	conditional chaos
$(b)_\top$	conditional magic
$\lhd b \rhd$	binary conditional
$\rtimes$	termination
$\ltimes$	non-termination
$\lfloor\, q\, \rfloor$	post-transition
$\lceil\, q\, \rceil$	pre-transition
P^n	iteration n times
P^*	arbitrary iteration
$P^{\circledast}$	finite iteration
P^+	non-skipped iteration
$P^{\oplus}$	non-skipped finite iteration
$\mid q \mid$	safety property
$\llbracket q \rrbracket$	terminating pre-post liveness property
$P \rightrightarrows Q$	rely-guarantee specifciation

Figure 3: Derived Logs commands

Definition 2 $$\begin{array}{rcl} \rtimes & \hat{=} & [\,\mathtt{true}\,]^{\circledast} \\ \ltimes & \hat{=} & [\,\mathtt{true}\,]^{\infty} \end{array}$$

A *post-transition* is a 1LOGS command in which the intermediate state is unchanged from the initial state; while a pre-transition is a 1LOGS command in which the final state is unchanged from the intermediate state. Both are useful in expressing safety and liveness properties. In the following definitions we assume q to be a predicate such that $q = q(\overleftarrow{w}, \overrightarrow{w})$.

Definition 3 $$\begin{array}{rcl} \lfloor\, q \,\rfloor & \hat{=} & [\, q \wedge \overleftarrow{w} = w \,] \\ \lceil\, q \,\rceil & \hat{=} & [\, q \wedge w = \overrightarrow{w} \,] \end{array}$$

P^n denotes the n-fold sequential iteration of P, where n can be any natural number or infinity in which case the iteration is the least fixed point of function $(\lambda X \cdot P\,;\,X)$. P^* denotes arbitrary iteration of P; $P^{\circledast}$ denotes finite iteration of P; P^+ denotes iteration at least once; and $P^{\oplus}$ denotes finite iteration at least once.

Definition 4 $$\begin{array}{rcl} P^0 & \hat{=} & \mathbb{I} \\ P^{n+1} & \hat{=} & P\,;\,P^n \\ P^{\infty} & \hat{=} & \bigsqcup_{n<\infty} (\, P^n\,;\,\perp\,) \\ P^* & \hat{=} & \bigsqcap_{0 \leqslant n \leqslant \infty} P^n \\ P^{\circledast} & \hat{=} & \bigsqcap_{0 \leqslant n < \infty} P^n \\ P^+ & \hat{=} & \bigsqcap_{0 < n \leqslant \infty} P^n \\ P^{\oplus} & \hat{=} & \bigsqcap_{0 < n < \infty} P^n \end{array}$$

$|\, q \,|$ describes a safety property true throughout its execution; it may or may not terminate. Its specification is the weakest command satisfying that safety property. $[\![\, q \,]\!]$ describes a terminating liveness property whose final state is related to its initial state by predicate q; it allows an arbitrary number of global synchronisations before termination. Such safety and liveness properties are conveniently expressed in terms of iteration as follows. We assume $q = q(\overleftarrow{w}, \overrightarrow{w})$.

Definition 5 $$\begin{array}{rcl} |\, q \,| & \hat{=} & \lfloor\, q \,\rfloor^* \\ [\![\, q \,]\!] & \hat{=} & \bigsqcap_{0 \leqslant n < \infty} \langle\, q \,\rangle_n \end{array}$$

A computation satisfies a rely-guarantee $P \rightrightarrows Q$ iff whenever P is satisfied, Q is guaranteed. Thus $P \rightrightarrows Q$ allows any computation that does not satisfy P, in which case it may not guarantee Q. This corresponds to the reply-guarantee s in TLA [17] and UNITY [2].

Definition 6 $\quad P \rightrightarrows Q \;=\; {\sim}P \sqcap Q$

In this paper we find no need to exploit the temporal operators 'must' $\Box$ and 'may' $\Diamond$ both readily defined in LOGS. Their definitions are:

$$\begin{aligned} \Diamond P \;&\hat{=}\; \rtimes\,;P\,;\bot \\ \Box P \;&\hat{=}\; \sim\Diamond{\sim}P\,. \end{aligned}$$

In the Temporal Logic of Actions [17], an action describes a single-step transition, while here P may contain more than one step. An advantage of our approach is that we now can easily specify and reason about the relations between several consecutive steps. In this paper we continue to use the right-hand sides above since they seem little more complicated than the left-hand abbreviations.

3.1 Laws of derived commands

Although $\bot$ and $\top$ are defined as extreme elements they can be expressed in terms of internal predicates:

Law 17
$$\begin{aligned} \bot \;&=\; \textstyle\prod_{n\leqslant\infty}\langle\, \mathtt{true}\,\rangle_n && (1)\\ \top \;&=\; \langle\, \mathtt{false}\,\rangle_n\,. && (2) \end{aligned}$$

They are mutual inverses. $\bot$ is the zero of non-deterministic choice and the unit of parallel composition, while $\top$ is the unit of non-deterministic choice, the zero of parallel composition and the zero of sequential composition. $\mathbb{I}$ is the unit of sequential composition.

Law 18
$$\begin{aligned} &{\sim}\bot \;=\; \top && (1)\\ &\bot \sqcap P \;=\; \bot && (2)\\ &\bot \sqcup P \;=\; P && (3)\\ &\top \sqcap P \;=\; P && (4)\\ &\top \sqcup P \;=\; \top && (5)\\ &\top\,;P \;=\; P\,;\top \;=\; \top && (6)\\ &\ltimes\,;P \;=\; \ltimes && (7)\\ &\mathbb{I}\,;P \;=\; P\,;\mathbb{I} \;=\; P && (8) \end{aligned}$$

Negation forms a complement.

Law 19
$$P \sqcup \sim P \;=\; \top \tag{1}$$
$$P \sqcap \sim P \;=\; \bot \tag{2}$$

Conditional chaos and conditional magic satisfy these straight-forward laws.

Law 20
$$(b)_\top \,;\, \langle\, p \,\rangle_n \;=\; \langle\, b(\overleftarrow{w}\,) \wedge p \,\rangle_n \tag{1}$$
$$\langle\, p \,\rangle_n \,;\, (b)_\top \;=\; \langle\, p \wedge b(\overrightarrow{w}\,) \,\rangle_n \tag{2}$$

The iteration operators satisfy these laws where $\lambda, \mu \in \{n, \infty, *, \circledast, +, \oplus\}$.

Law 21
$$P^* = P^\circledast \sqcap P^\infty \tag{1}$$
$$P^+ = P^\oplus \sqcap P^\infty \tag{2}$$
$$P^\lambda \,;\, P^\mu = P^\mu \,;\, P^\lambda \tag{3}$$
$$P^* \,;\, P^* = P^\circledast \,;\, P^* = P^* \tag{4}$$
$$P^\circledast \,;\, P^\circledast = P^\circledast \tag{5}$$
$$P^\infty = P^\infty \,;\, \bot = P^\infty \,;\, P^\lambda \tag{6}$$
$$(P \sqcap Q)^\infty = P^* \,;\, Q \,;\, (P \sqcap Q)^\infty \tag{7}$$
$$P^\oplus \,;\, P^\circledast = P^\oplus \tag{8}$$
$$P^+ \,;\, P^* = P^+ \,;\, P^\circledast = P^\oplus \,;\, P^* = P^+ \tag{9}$$

An always-true safety property p distributes various structures consisting of post-transitions. In the following laws $\lambda\!:\!\{n, \infty, *, \circledast, +, \oplus\}$ and we assume that $P = P \sqcup \lvert\, \texttt{true} \,\rvert$ and $Q = Q \sqcup \lvert\, \texttt{true} \,\rvert$.

Law 22
$$\lvert\, p \,\rvert \sqcup \rtimes = \lfloor\, p \,\rfloor^\circledast \tag{1}$$
$$\lvert\, p \,\rvert \sqcup \langle\, q \,\rangle_0 = \mathbb{I} \sqcup \langle\, q \,\rangle_0 \tag{2}$$
$$\lvert\, p \,\rvert \sqcup \lfloor\, q \,\rfloor = \lfloor\, p \wedge q \,\rfloor \tag{3}$$
$$\lvert\, p \,\rvert \sqcup (\, P \,;\, Q \,) = \lvert\, p \,\rvert \sqcup P \,;\, \lvert\, p \,\rvert \sqcup Q \tag{4}$$
$$\lvert\, p \,\rvert \sqcup Q^\lambda = (\, \lvert\, p \,\rvert \sqcup Q \,)^\lambda \tag{5}$$
$$(\, \rtimes \,;\, \lvert\, p \,\rvert \,) \sqcup (\, \rtimes \,;\, Q^\infty \,) = (\, \rtimes \,;\, \lvert\, p \,\rvert \sqcup Q^\infty \,) \tag{6}$$

The parallel composition of pre-post s becomes a conjunction of their internal predicates, while a non-deterministic choice becomes a disjunction.

Law 23
$$[\![\, p \,]\!] \sqcup [\![\, q \,]\!] \;=\; [\![\, p \wedge q \,]\!] \tag{1}$$
$$[\![\, p \,]\!] \sqcap [\![\, q \,]\!] \;=\; [\![\, p \vee q \,]\!] \tag{2}$$

Reactive commands satisfy the laws:

Law 24
$$\rtimes\,;\,\rtimes \;=\; \rtimes \tag{1}$$
$$\rtimes^{\oplus} = \rtimes^{\circledast} = \rtimes \tag{2}$$
$$\langle\, p\,\rangle_1^{\circledast} \sqcap \rtimes = \rtimes \tag{3}$$
$$(P \sqcup \ltimes)\,;\,Q = P \sqcup \ltimes \tag{4}$$
$$\rtimes\,;\,\lfloor\, p(\overleftarrow{w})\,\rfloor^{\infty} = \rtimes\,;\,\lfloor\, p(\overrightarrow{w})\,\rfloor^{\infty} \tag{5}$$
$$\rtimes\,;\,\lfloor\, p\,\rfloor^{\infty} \sqcup \rtimes\,;\,\lfloor\, q\,\rfloor^{\infty} = \rtimes\,;\,\lfloor\, p \wedge q\,\rfloor^{\infty} \tag{6}$$
$$(\,\rtimes\,;\,P\,) \sqcup (\,\rtimes\,;\,Q\,) = \rtimes\,;\,(\,(\rtimes\,;\,P) \sqcup Q\,) \sqcap (\,P \sqcup (\rtimes\,;\,Q)\,) \tag{7}$$

A rely-guarantee satisfies the laws, motivated by propositional laws:

Law 25
$$P \rightrightarrows P \;=\; \bot \tag{1}$$
$$P \sqcup (\,P \rightrightarrows Q\,) \;=\; P \sqcup Q \tag{2}$$

4 Case study: the dining-philosophers

4.1 Dining philosophers

Since it was first described in [9], the example of the dining philosophers has become a benchmark for the calibration of theories of concurrency and the way they facilitate reasoning about resource contention. Five philosophers are seated at a circular dining table. Each philosopher cycles through the phases of *thinking*, t, being *hungry*, h, and *eating*, e. Neighbouring philosophers may not eat at the same time. We require that a hungry philosopher eventually eats provided that thinking and eating are achieved in finitely-many steps.

Let each philosopher have state $x_k : \{t, h, e\}$ and let the state of the vector of philosophers be $x \mathrel{\hat{=}} x_0, x_1, \ldots, x_4$. The initial state of the system of philosophers is specified as follows, with $k^- \mathrel{\hat{=}} (k{-}1)\ \mathit{mod}\ 5$ and $k^+ \mathrel{\hat{=}} (k{+}1)\ \mathit{mod}\ 5$.

$$\textbf{SPEC 1} \quad \mathrel{\hat{=}} \quad \bigsqcup_k \left(\begin{array}{l} \lfloor\, \overleftarrow{x}_k = t\,\rfloor^{\oplus}\,; \\ \lfloor\, \overleftarrow{x}_k = h\,\rfloor^{\oplus}\,; \\ \lfloor\, \overleftarrow{x}_{k^-} \neq \overleftarrow{x}_k = e \neq \overleftarrow{x}_{k^+}\,\rfloor^{\oplus} \end{array} \right)^{\infty}$$

The term $\lfloor\, \overleftarrow{x}_k = h\,\rfloor^{\oplus}$, for example, specifies termination of philosopher k's hungry phase.

4.2 Forks

The specification is to be refined by a distributed design in which contention is mediated by forks, one between adjacent philosophers and numbered like the philosophers: philosopher k requires forks k and k^+ in order to eat. Let fork k be denoted y_k with the vector of forks being denoted $y \mathrel{\hat=} y_0, y_1, \ldots, y_4$. Each fork has two states: either l (being used by the philosopher to its left) or r (to its right). Thus to eat, philosopher k requires $y_k = r$ and $y_{k^+} = l$.

$$\textbf{SPEC 2} \quad \mathrel{\hat=} \quad \bigsqcup_k \begin{pmatrix} \lfloor \overleftarrow{x}_k = t \rfloor^{\oplus} ; \\ \lfloor \overleftarrow{x}_k = h \rfloor^{\oplus} ; \\ \lfloor \overleftarrow{x}_k = e \rfloor^{\oplus} \end{pmatrix}^{\infty} \sqcup \lfloor \mathit{forks} \rfloor$$

$$\textit{where} \;\; \mathit{forks} \;\mathrel{\hat=}\; \forall k \cdot (\overleftarrow{x}_k = e \;\Rightarrow\; \overrightarrow{y}_k = r \wedge \overrightarrow{y}_{k^+} = l)$$

The validity of that design is ensured as follows.

Theorem 2 SPEC1 $\sqsubseteq$ SPEC2

Proof

$$\begin{array}{ll} \text{SPEC2} & \\ = & \text{law } \mathbf{11}\text{, law } \mathbf{22}(5)\text{, law } \mathbf{22}(4) \text{ and law } \mathbf{22}(3) \\ \bigsqcup_k \begin{pmatrix} \lfloor \overleftarrow{x}_k = t \wedge \mathit{forks} \rfloor^{\oplus} ; \\ \lfloor \overleftarrow{x}_k = h \wedge \mathit{forks} \rfloor^{\oplus} ; \\ \lfloor \overleftarrow{x}_k = e \wedge \overrightarrow{y}_k = r \wedge \overrightarrow{y}_{k^+} = l \wedge \mathit{forks} \rfloor^{\oplus} \end{pmatrix}^{\infty} & \\ \sqsupseteq & \text{monotonicity of various compositions} \\ \text{SPEC1} \;. & \end{array}$$

□

4.3 A strategy

In this design the forks are resources and the philosophers are (resource) consumers. The thinking phase represents a period during which a consumer needs no shared resource; the hungry phase represents a period of waiting for required resources; the eating phase represents a resource-consuming period. Thus termination of both thinking and eating must be guaranteed by each resource consumer, whilst termination of the hungry phase has to be guaranteed by a distributed implementation.

Initially we suppose that termination of the hungry phase is unknown. We propose the following strategy and later prove it to terminate:

> a thinking philosopher may either continue thinking or become hungry (termination of the thinking phase is specified by finiteness of its iteration);
>
> a hungry philosopher may either remain hungry or immediately eat, provided two adjacent forks are available (termination is unknown at this stage);
>
> an eating philosopher may either continue eating or stop to think (however the eating phase terminates);
>
> if two adjacent philosophers are thinking, the fork between them will not change state;
>
> if a philosopher is thinking while his left-hand neighbour is not, then the philosopher will 'lose' the fork between them;
>
> if a philosopher is not thinking but his left-hand neighbour is, the neighbour will lose the fork between them;
>
> if two adjacent philosophers are hungry, the fork between them will not change state;
>
> an eating philosopher will 'obtain' both neighbouring forks.

That strategy is formalised as follows.

SPEC 3 $\quad \hat{=} \quad (acyc)_{\top} \,;\, \bigsqcup_k P_k$

where

$$
\begin{array}{rcl}
P_k & \hat{=} & \left(T_k^{\oplus}\,;\, H_k^{+}\,;\, E_k^{\oplus}\right)^{\infty} \\
T_k & \hat{=} & \lfloor \overleftarrow{x}_k = t \wedge stgy_k \rfloor \\
H_k & \hat{=} & \lfloor \overleftarrow{x}_k = h \wedge stgy_k \rfloor \\
E_k & \hat{=} & \lfloor \overleftarrow{x}_k = e \wedge stgy_k \rfloor \\
stgy_k & \hat{=} & \quad (\overleftarrow{x}_k = t) \Rightarrow (\overrightarrow{x}_k = \overleftarrow{x}_k \sqcap h) \\
 & & \wedge\, (\overleftarrow{x}_k = h) \Rightarrow (\overrightarrow{x}_k = e \lhd \overleftarrow{y}_k = r \wedge \overleftarrow{y}_{k+} = l \rhd \overleftarrow{x}_k) \\
 & & \wedge\, (\overleftarrow{x}_k = e) \Rightarrow (\overrightarrow{x}_k = \overleftarrow{x}_k \sqcap t) \\
 & & \wedge\, (\overleftarrow{x}_{k-} = t = \overleftarrow{x}_k) \Rightarrow (\overleftarrow{y}_k = \overrightarrow{y}_k) \\
 & & \wedge\, (\overleftarrow{x}_{k-} \neq t = \overleftarrow{x}_k) \Rightarrow (\overrightarrow{y}_k = l) \\
 & & \wedge\, (\overleftarrow{x}_{k-} = t \neq \overleftarrow{x}_k) \Rightarrow (\overrightarrow{y}_k = r) \\
 & & \wedge\, (\overleftarrow{x}_{k-} = h = \overleftarrow{x}_k) \Rightarrow (\overleftarrow{y}_k = \overrightarrow{y}_k) \\
 & & \wedge\, (\overleftarrow{x}_{k-} = e) \Rightarrow (\overrightarrow{y}_k = l) \\
 & & \wedge\, (\overleftarrow{x}_k = e) \Rightarrow (\overrightarrow{y}_k = r) \\
acyc & \hat{=} & \neg(y_0 = y_1 = \cdots = y_4)
\end{array}
$$

P_k represents philosopher k, while T_k, H_k and E_k represent thinking, hungry and eating, respectively. SPEC3 is initialised by the requirement, *acyc*, that the forks form an *acyclic* priority graph.

The following lemma states that the fork graph is always acyclic. (In the case of improper initialisation, SPEC3 would become miraculous and so still refine SPEC2.)

Lemma 3 (acyclic safety) $\quad \mathrm{SPEC}\,3 \;\sqsupseteq\; \lfloor \overleftarrow{acyc} \rfloor^{\infty}$

Proof Let $strategy \;\hat{=}\; \bigwedge_k stgy_k$. Then, by law 12 and monotonicity of $\lfloor \cdot \rfloor^{\infty}$,

$$\mathrm{SPEC3} \;\sqsupseteq\; (acyc)_{\top}\,;\, \lfloor strategy \rfloor^{\infty} \;.$$

From monotonicity of $\lfloor \cdot \rfloor$ it follows, by considering all possible types of step, that $\overleftarrow{acyc}$; indeed:

1. if all philosophers are thinking then

   $\lfloor strategy \wedge \overleftarrow{x} = t,t,t,t,t \rfloor \;\sqsupseteq\; \lfloor \overleftarrow{y} = \overrightarrow{y} \rfloor \;\sqsupseteq\; \lfloor \overleftarrow{acyc} \Rightarrow \overrightarrow{acyc} \rfloor$;

2. if a philosopher k is eating then

   $\lfloor strategy \wedge \overleftarrow{x}_k = e \rfloor \;\sqsupseteq\; \lfloor \overrightarrow{y}_k \neq \overrightarrow{y}_{k+} \rfloor \;\sqsupseteq\; \lfloor \overrightarrow{acyc} \rfloor$;

3. if all philosophers are hungry then

   $$\lfloor\, strategy \wedge \overleftarrow{x} = h,h,h,h,h \,\rfloor \;\sqsupseteq\; \lfloor\, \overleftarrow{y} = \overrightarrow{y} \,\rfloor \;\sqsupseteq\; \lfloor\, \overleftarrow{acyc} \Rightarrow \overrightarrow{acyc} \,\rfloor;$$

4. if philosophers i to j are thinking while philosophers i^- and j^+ are hungry then

   $$\begin{array}{l} \lfloor\, strategy \wedge (\overleftarrow{x}_i = \overleftarrow{x}_{i^+} = \cdots = \overleftarrow{x}_j = t) \wedge (\overleftarrow{x}_{i^-} = \overleftarrow{x}_{j^+} = h) \,\rfloor \\ \sqsupseteq \\ \lfloor\, \overrightarrow{y}_i \neq \overrightarrow{y}_{j^+} \,\rfloor \\ \sqsupseteq \\ \lfloor\, \overrightarrow{acyc} \,\rfloor \;\; . \end{array}$$

Thus

$$\begin{array}{ll} \text{SPEC3} & \\ \sqsupseteq & \\ (acyc)_\top ;\ \lfloor\, strategy \,\rfloor^\infty & \\ \sqsupseteq & \text{law } \mathbf{2} \\ (acyc)_\top ;\ \lfloor\, \overrightarrow{acyc} \vee (\, \overleftarrow{acyc} \Rightarrow \overrightarrow{acyc} \,) \,\rfloor^\infty & \\ \sqsupseteq & \text{monotonicity of } \lfloor \cdot \rfloor^\infty \\ (acyc)_\top ;\ \lfloor\, \overleftarrow{acyc} \Rightarrow \overrightarrow{acyc} \,\rfloor^\infty & \\ \sqsupseteq & \text{law } \mathbf{20}(1), \text{ law } \mathbf{20}(2) \text{ and definition of } \lfloor \cdot \rfloor^\infty \\ \lfloor\, \overleftarrow{acyc} \,\rfloor^\infty \;\; . & \end{array}$$

□

4.4 Liveness

To reason about liveness we first decompose each philosopher k in two:

$$\begin{array}{lcl} L_k & \hat{=} & \left(T_k^\oplus \,;\, H_k^\oplus \,;\, E_k^\oplus \right)^\infty \\ D_k & \hat{=} & \left(T_k^\oplus \,;\, H_k^\oplus \,;\, E_k^\oplus \right)^\circledast \,;\, T_k^\oplus \,;\, H_k^\infty \;\; . \end{array}$$

We think of L_k as denoting a 'living' philosopher and D_k as denoting a 'dying' one who eventually remains hungry forever.

For each philosopher life or death is a non-deterministic choice! Indeed:

$$
\begin{aligned}
& P_k \\
& = \qquad \text{definition of } P_k \\
& \left(T_k^{\oplus} \,;\, H_k^{+} \,;\, E_k^{\oplus} \right)^{\infty} \\
& = \qquad \text{law } \mathbf{21}(1) \\
& \left(T_k^{\oplus} \,;\, (H_k^{\oplus} \sqcap H_k^{\infty}) \,;\, E_k^{\oplus} \right)^{\infty} \\
& = \qquad \text{law } \mathbf{9}\text{, law } \mathbf{7}\text{, law } \mathbf{18}(7) \text{ and definition of } \lfloor \cdot \rfloor^{\infty} \\
& \left(T_k^{\oplus} \,;\, H_k^{\oplus} \,;\, E_k^{\oplus} \right)^{*} \,;\, T_k^{\oplus} \,;\, H_k^{\infty} \\
& = \qquad \text{law } \mathbf{21}(7)\text{, law } \mathbf{21}(6) \text{ and law } \mathbf{18}(7) \\
& \Big(\left(T_k^{\oplus} \,;\, H_k^{\oplus} \,;\, E_k^{\oplus} \right)^{\circledast} \,;\, T_k^{\oplus} \,;\, H_k^{\infty} \Big) \sqcap \left(T_k^{\oplus} \,;\, H_k^{\oplus} \,;\, E_k^{\oplus} \right)^{\infty} \\
& = \qquad \text{definitions of } D_k \text{ and } L_k \\
& D_k \sqcap L_k \ .
\end{aligned}
$$

Now we observe that SPEC3 is refined by a *group* of living philosophers:

SPEC 4 $\quad \widehat{=} \quad (acyc)_{\top} \,;\, \bigsqcup_k L_k$

In fact SPEC4 equals SPEC3 because the strategy guarantees that every philosopher lives. To prove that we need to eliminate the possibility of deadlock or starvation. Lemma 4 excludes the possibility that any philosopher dies by waiting for a fork from a neighbour (i.e. deadlock), whilst lemma 6 excludes that of starvation.

Lemma 4 (deadlock freedom) $\quad (acyc)_{\top} \,;\, \bigsqcup_k D_k \ = \ \top$

Proof

$$
\begin{aligned}
& (acyc)_{\top} \,;\, \bigsqcup_k D_k \\
& \sqsupseteq \qquad \text{monotonicty of } \lfloor \cdot \rfloor^{\infty} \text{ and law } \mathbf{11} \\
& (\, (acyc)_{\top} \,;\, \lfloor\, strategy \,\rfloor^{\infty} \,) \sqcup \bigsqcup_k (\, \rtimes \,;\, \lfloor\, \overleftarrow{x}_k = h \,\rfloor^{\infty} \,) \\
& \sqsupseteq \qquad \text{law } \mathbf{20}(1)\text{, law } \mathbf{20}(2)\text{, definition of } \lfloor \cdot \rfloor^{\infty}\text{, law } \mathbf{11} \text{ and law } \mathbf{24}(6) \\
& \lfloor\, strategy \wedge \overleftarrow{acyc} \,\rfloor^{\infty} \sqcup (\, \rtimes \,;\, \lfloor\, \overleftarrow{x} = h, h, h, h, h \,\rfloor^{\infty} \,) \\
& \sqsupseteq \qquad \text{law } \mathbf{22}(4)\text{, law } \mathbf{22}(5)\text{, law } \mathbf{22}(3) \text{ and monotonicity of } \lfloor \cdot \rfloor^{\infty} \\
& (\, \rtimes \,;\, \lfloor\, \overrightarrow{x} \neq h, h, h, h, h \,\rfloor^{\infty} \,) \sqcup (\, \rtimes \,;\, \lfloor\, \overleftarrow{x} = h, h, h, h, h \,\rfloor^{\infty} \,)
\end{aligned}
$$

$=$ law **24**(6), law **24**(5), definition of $\top$ and law **18**(6)

$\top$.

$\square$

The following lemma shows that a living philosopher is unable to dine with a dying neighbour.

Lemma 5 $\quad L_{k^-} \sqcup D_k \;=\; \top$

Proof

$$L_{k^-} \sqcup D_k$$

$\sqsupseteq$ $\rtimes$ is the weakest terminating command

$$(\rtimes\,;\, L_{k^-}) \sqcup (\rtimes\,;\, H_k^{\infty})$$

$\sqsupseteq$ weakening infinite iteration with definition of $\lfloor \cdot \rfloor$

$$(\rtimes\,;\, L_{k^-}) \sqcup (\rtimes\,;\, H_k^{*})$$

$=$ law **22**(6), law **22**(4), law **22**(5) and law **22**(3)

$$\rtimes\,;\, (\,(T_{k^-} \sqcup H_k)^{\oplus}\,;\, (H_{k^-} \sqcup H_k)^{\oplus}\,;\, (E_{k^-} \sqcup H_k)^{\oplus}\,)^{\infty}$$

$\sqsupseteq$ monotonicty of $\lfloor \cdot \rfloor^{\lambda}$

$$\begin{aligned}\rtimes\,;\, (\,&\lfloor \overrightarrow{y}_k = r \rfloor^{\oplus}\,;\\ &\lfloor (\overleftarrow{x}_{k^-} = h) \wedge (\overleftarrow{y}_k = \overrightarrow{y}_k) \wedge (\overleftarrow{y}_k = r \Rightarrow \overleftarrow{x}_{k^-} = \overrightarrow{x}_{k^-}) \rfloor^{\oplus}\,;\\ &\lfloor \overleftarrow{x}_{k^-} = e \rfloor^{\oplus}\,)^{\infty}\end{aligned}$$

$\sqsupseteq$ monotonicty of $\lfloor \cdot \rfloor^{\lambda}$

$$\rtimes\,;\, (\,\lfloor \overrightarrow{y}_k = r \rfloor^{\oplus}\,;\, \lfloor \overrightarrow{x}_{k^-} = h \rfloor^{\oplus}\,;\, \lfloor \overleftarrow{x}_{k^-} = e \rfloor^{\oplus}\,)^{\infty}$$

$\sqsupseteq$ monotonicty of $\lfloor \cdot \rfloor^{\lambda}$, law **20**(1) and law **20**(2)

$$\rtimes\,;\, (\,\rtimes\,;\, \lfloor \overrightarrow{x}_{k^-} = h \rfloor^{\oplus}\,;\, (x_{k^-} = h)_{\top}\,;\, (x_{k^-} = e)_{\top}\,;\, \lfloor x_{k^-} = e \rfloor^{\oplus}\,)^{\infty}$$

$=$ law **20**(3), law **18**(4) and law **11**(2)

$\top$.

$\square$

We infer that at least one philosopher is living and no living philosopher can sit besides a dying neighbour. Consequently all philosophers are living.

Lemma 6 (starvation freedom) SPEC3 = SPEC4

Proof

$$
\begin{array}{ll}
\text{SPEC3} & \\
\;= & \text{definition of } P_k \\
(acyc)_\top \,;\, \bigsqcup_k (L_k \sqcap D_k) & \\
\;= & \text{law } \mathbf{1} \\
\begin{array}{l} (acyc)_\top \,;\, \bigsqcup_k L_k \sqcap (D_0 \sqcup L_1 \sqcup \cdots \sqcup L_4) \\ \qquad\qquad\qquad \sqcap \cdots\cdots \\ \qquad\qquad\qquad \sqcap (L_0 \sqcup D_1 \sqcup \cdots \sqcup D_4) \\ \qquad\qquad\qquad \sqcap \bigsqcup_k D_k \end{array} & \\
\;= & \text{lemma 4 and lemma 5} \\
((acyc)_\top \,;\, \bigsqcup_k L_k \sqcap \top) \sqcap \top & \\
\;= & \text{law } \mathbf{18}(4) \\
acyc_\top \,;\, \bigsqcup_k L_k & \\
\;= & \\
\text{SPEC4}\ . &
\end{array}
$$

$\square$

In fact specification SPEC4 is very close to a BSP program. For details of such a program and its derivation from SPEC4 we refer to [4]. Here we are interested simply in validity of SPEC3 with respect to SPEC1:

Theorem 3 SPEC2 $\sqsubseteq$ SPEC3

Proof According to lemma 6 we need check only that SPEC2 $\sqsubseteq$ SPEC4. But

$$
\begin{array}{ll}
\text{SPEC4} & \\
\;\sqsupseteq & \text{weakening} \\
\lfloor\, strategy \,\rfloor^\infty & \\
\;\sqsupseteq & \text{weakening}
\end{array}
$$

$$\lfloor \forall k \cdot (x_k = e \Rightarrow y_k = r) \wedge (x_{k^-} = e \Rightarrow y_k = l) \rfloor^\infty$$

$$=$$

$$\lfloor \forall k \cdot x_k = e \;\Rightarrow\; (y_k = r \wedge y_{k^+} = l) \rfloor^\infty$$

$$\sqsupseteq \qquad \text{weakening infinite iteration}$$

$$|\,\mathit{forks}\,| \;.$$

$\square$

5 Semantics of Logs

In this section we present a predicate semantics for Logs with respect to which the laws provided in this paper are sound, and exhibit healthiness conditions which characterise Logs commands. Proofs and further details appear in [4].

5.1 Healthiness conditions

Let $\omega : \Omega$ denote the vector of all program (or system) variables. For each logical variable x in ω its initial state is denoted x, its final state is denoted x', its initial trace record is denoted $x.tr$, and its final trace record is denoted $x.tr'$ $(= (x.tr)')$. For brevity we adopt these conventions:

> ω' denotes the vector of dashed variables x' for x in ω
>
> τ denotes the vector of trace variables $x.tr$ for x in ω
>
> τ' denotes the vector of trace variables $x.tr'$ for x in ω.

From section 2.1 we recall that a Logs command can be expressed as a predicate with free variables $\omega, \omega', \tau, \tau'$. However not every such predicate represents a Logs command. The following four *healthiness conditions*, presented here in the style of [15], must hold. (For a detailed description of healthiness conditions in this context we refer to [6]. The conditions here arise from a more general unifying theory called *cumulative computing* [4].)

The first healthiness condition states that if a command starts after a failure or a non-terminating command, it becomes unobservable: if the initial trace record is infinite then the predicate is arbitrary. Writing $|\tau|$ for the length of sequence τ, that is expressed:

Healthiness condition: $H_0(A) \ \hat{=}\ A = A \vee |\tau| = \infty$

The trace record of Logs is monotonic: a command retains the trace before its start but may append to it. Writing $\tau \leq \tau'$ to mean that sequence τ is a prefix of sequence τ', that is expressed:

Healthiness condition: $H_\tau(A) \ \hat{=}\ A = A \wedge \tau \leq \tau'$

The final state of a non-terminating Logs command is not observable.

Healthiness condition: $H_\infty(A) \ \hat{=}\ A = A \vee A\,;(|\tau'| = \infty \wedge \tau \leq \tau')$

A Logs command does not depend on the trace record before its start and satisfies an additional healthiness condition called *shift invariance*: a Logs command can be 'moved' backwards or forwards without change. Note that a non-terminating command with an unobservable final state cannot, however, be moved backwards to become a terminating command!

Healthiness condition: $H_\leftrightarrow(A) \ \hat{=}\ A = \exists \tau_0, \tau_0' \cdot A[\tau_0, \tau_0'/\tau, \tau'] \wedge (\tau_0' - \tau_0 = \tau' - \tau) \wedge (|\tau_0'| = \infty \Rightarrow |\tau'| = \infty)$

In fact $H_\leftrightarrow$ can be written as a generic composition and so *each* of the healthiness functions is monotonic and idempotent on predicates (see [4] for details). Thus the composed function

$$H \ \hat{=}\ H_\leftrightarrow \circ H_\infty \circ H_\tau \circ H_0$$

is again monotonic and idempotent which, by the inheritance theorem in [4], ensures:

(a) the range of H forms a complete lattice (consisting of the 'healthy predicates') on which the combinators $\sqcap$, $\sqcup$, (;) and `var` v are closed, and

(b) the laws of that complete lattice involving only $\sqcap$, $\sqcup$ and (;) are inherited from the more abstract language CML+ (see [4]) so that the soundness of the foregoing laws of Logs is assured.

We note that conversely every healthy predicate forms the predicative semantics of a LOGS command; see [4].

5.2 Predicative semantics

We refer to section 2.1 for the intuition behind a basic LOGS command, and define its predicative semantics as follows.

The semantics of a nLOGS command is simply the predicate $P(\omega, \omega', \tau, \tau')$ which describes its effect in terms of those free variables. Examples have appeared in section 4. From now on we assume each predicate has those four vectors of free variables.

The semantics of $P \sqcap Q$ is the disjunction of the semantics of P and Q. This corresponds to the usual representation of the demonic nondeterminism arising from the abstraction resulting from concealment of local variables.

The semantics of $P\,;\,Q$ for P is given, as usuall, by relational composition

$$P\,;\,Q \;\hat{=}\; \exists \omega_0, \tau_0 \cdot P[\omega_0, \tau_0/\omega', \tau'] \wedge Q[\omega_0, \tau_0/\omega, \tau]\,.$$

In particular that yields law 5; and if P is non-terminating then $P\,;\,Q$ equals P.

The semantics of $P \sqcup Q$ is the conjunction of the semantics of P and Q. That provides the simplest kind of parallel composition, and no real parallel computer system directly implements it. Parallel compositions in real programming languages are normally much more complicated; their implementation details are mainly due to efficiency concerns and not suitable for specification purposes, which is of primary concern here.

The semantics of `var` v P is not simply the existential quantification of the relevant variables in P since that would completely hide $v, v', v.tr, v.tr'$ which ought at least to satisfy the healthiness conditions. So we need to close that under the healthiness function H. The result is $H(\exists \omega, \omega', \tau, \tau' \cdot P)$.

The semantics of $\sim P$ is the closure under H of the negation of the predicative semantics of P.

The semantics of derived LOGS commands follow from their definitions, provided in section 3, in terms of the basic commands.

6 Conclusion

BSP programming can be considered distributed programming constrained by the requirement that non-determinism be bounded. Such a constraint corresponds directly to SIMD data parallelism and also substantially simplifies many MIMD distributed algorithms. However, partially as a result of that, BSP is lower-level than might be desired when it comes to reasoning about (for example when deriving!) an implementation. Further limitations of BSP have been its lack of reactive computations, lack of task parallelism, and lack of unbounded nondeterminism.

LOGS integrates the pre-post, finite and reactive-process styles of specification, as indicated by its laws and by our study of the (nonterminating, reactive) dining philosophers. Its refinement and solution are encouragingly simple compared with those in say UNITY [2]. This is partly because of the simplicity of LOGS and partly because of the help provided by sequential composition in reasoning about liveness. Use of logical conjunction as abstract parallel composition has also proved convenient, as has been the definition of rely-guarantee specification analogous to logical implication.

We have thus made the case for a simple but comprehensive and expressive specification language to fill the gap in abstraction suffered by BSP. It continues to support both SIMD and MIMD applications. LOGS has been designed for use at the intermediate stages of program derivation, with the more complicated features of BSP being added only towards the end of the derivation. Once the derivation reaches a sufficiently low-level, a complete set of laws is now available to reach a BSP implementation. In [4] several languages with different levels of abstraction are introduced for BSP. An abstract specification is first written in LOGS thought of as a specification language. It is then decomposed into smaller specifications containing no more than one synchronisation. Refinement laws are used to transform the resulting simple LOGS specifications to BSP programs with variable protection. Variable protections can then moved out of parallel compositions. Finally if there are no further BSP process in parallel, variable protections outside all parallel compositions are eliminated.

That derivation methodology differs substantially from previous work on BSP which has focused on how to reduce BSP to data-parallelism and how to apply the well-established sequential refinement calculus to it. In the standard methodology a specification is first decomposed into bulk-synchronous supersteps. Each step is then treated as a vector of similar sequential programs associated with different data on different processors [22]. Our approach does not practise that simplification, but fully supports MIMD parallel programming, frequently considered infeasible in practical programming.

One particularly pleasing feature of the work has been the successful use of the technique of inheritance. For further details concerning the inheritance relations between semantic models based on healthiness conditions, and for application of the technique to unification of real-time, reactive processes and cost models, we refer to [4] where the predicative semantics of LOGS is naturally derived from that theoretic framework.

Important work remaining includes extension of these techniques to handle more-involved forms of parallel composition, and investigation of the link between temporal logic and LOGS.

References

[1] R. Back and K. Sere, Stepwise refinement of action systems. *Lecture Notes in Computer Science*, **375**:115–138, Springer-Verlag, 1989.

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

[3] Y. Chen, Specification for reactive bulk-synchronous programming. In *Proc. 8th Euromicro Workshop on Parallel and Distributed Processing*, 190–196, IEEE Computer Society Press, 2000.

[4] Y. Chen, *Formal Methods for Global Synchrony.* DPhil. Thesis, Oxford University Computing Laboratory, to appear.

[5] Y. Chen, How to write a healthiness condition. In *Second International Conference on Integrated Formal Methods*, ed. W.Grieskamp, T. Santen and W. Stoddart, Lecture Notes in Computer Science **1945**:299–317, Springer-Verlag, 2000.

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

[7] Y. Chen and J. W. Sanders, The weakest specifunction. Oxford University Computing Laboratory, Programming Research Group, technical report TR-8-00, 2000.

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

[9] E. W. Dijkstra, Two starvation free solutions to the general exclusion problem. EWD **625**, 1978.

[10] J. He, Q. Miller, L. Chen, Algebraic Laws for BSP Programming. Euro-Par'96, *Lecture Notes in Computer Science*, **1124**:359–367, 1996.

[11] E. C. R. Hehner, Predicative Programming I, II. *Communications of ACM*, **27**(2):134–143,144–151, 1984.

[12] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas and R. Bisseling, BSPlib: the BSP programming library. *Parallel Computing*, **24**(14):1947–1980, 1998.

[13] C. A. R. Hoare, Programs are predicates. In *Mathematical Logic and Programming Languages*, ed. J. C. Shepherdson and C. A. R. Hoare, 141–155, Prentice Hall series in Computer Science, 1985.

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

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

[16] B. von Karger, A calculational approach to reactive systems. *Science of Computer Programming*, **37**:139–161, 2000.

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

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

[19] R. D. Maddux, Relation-algebraic semantics. *Theoretical Computer Science*, **160**:1–85, 1996.

[20] Carroll Morgan, *Programming from Specifications*, second edition. Prentice Hall International, 1994.

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

[22] D. B. Skillicorn, Building BSP programs using the refinement calculus. *Formal Methods for Parallel Programming and Applications*, IPPS/SPDP'98, 1998.

[23] D. B. Skillicorn, J. M. D. Hill and W. F. McColl, Questions and answers about BSP. *Scientific Programming*, **6**(3):249–274, 1997.

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

[25] C. Zhou, C. A. R. Hoare and A. P. Ravn, A calculus of durations. *Information Processing Letters*, **40**(5):269–276, 1992.