Electronic Notes in Theoretical Computer Science 68 No. 2(2002)
URL: http://www.elsevier.nl/locate/entcs/volume68.html 18 pages

A Logical Process Calculus [1]

Rance Cleaveland [2]

Department of Computer Science, State Univ. of New York at Stony Brook, USA

Gerald Lüttgen [3]

Department of Computer Science, Univ. of Sheffield, U.K.

Abstract

This paper presents the Logical Process Calculus (LPC), a formalism that supports *heterogeneous* system specifications containing both operational and declarative subspecifications. Syntactically, LPC extends Milner's Calculus of Communicating Systems with operators from the alternation–free linear–time μ–calculus (LTμ). Semantically, LPC is equipped with a behavioral preorder that generalizes Hennessy's and De Nicola's must–testing preorder as well as LTμ's satisfaction relation, while being compositional for all LPC operators. From a technical point of view, the new calculus is distinguished by the inclusion of (i) both minimal and maximal fixed–point operators and (ii) an unimplementability predicate on process terms which tags inconsistent specifications. The utility of LPC is demonstrated by means of an example highlighting the benefits of heterogeneous system specification.

1 Introduction

Over the past two decades, a wealth of approaches to formally specifying and reasoning about reactive systems have been introduced. Most of these may be classified according to whether they are based on *process algebra* [3] or *temporal logic* [28]. The process–algebraic paradigm is founded on notions of *refinement*, where one typically formulates a system specification and its implementation in the same notation and then proves that the latter refines the former. The underling semantics is usually given operationally, and refinement relations are formalized as preorders. In contrast, the temporal–logic paradigm is based on the use of *temporal logics* [28] to formulate specifications,

[1] Research support was provided under NASA Contract No. NAS1–97046 and by NSF grant CCR–9988489. The first author was also supported by AFOSR Grant F49620–95–1–0508, ARO Grant P–38682–MA, and NSF Grants CCR–9505562, CCR–9996086, and INT–9996095.

[2] Email: rance@cs.sunysb.edu.

[3] Email: g.luettgen@dcs.shef.ac.uk.

©2002 Published by Elsevier Science B.V.

with implementations being given in an operational notation. One then verifies a system by establishing that it is a *model* of its specification, in the formal logical sense. The strength of the former paradigm is its support for *compositional reasoning*, i.e., one may refine system components independently of others. The benefit of the latter paradigm originates in its support for abstract specifications, where irrelevant operational details may be ignored. Both approaches may be given automated support in the form of *model checking* when the considered systems are finite–state.

The objective of this paper is to develop a *compositional theory for heterogeneous specifications* that uniformly integrates both refinement–based and temporal–logic specification styles, thereby allowing both approaches to be taken advantage of when designing systems. Accordingly, we present a novel *Logical Process Calculus* (LPC) that combines the algebraic operators of Milner's *Calculus of Communicating Systems* (CCS) [26] with the logical operators of the *Alternation–Free Linear–Time μ–Calculus* (LTμ) [33]. More precisely, we show that logical disjunction in LTμ may be understood as internal choice, complementing the external choice operator in CCS, and logical conjunction in LTμ as synchronous parallel composition, complementing asynchronous parallel composition in CCS. Moreover, LTμ is equipped with two recursion operators, a least fixed–point operator and a greatest fixed–point operator, which allow for the finite but unbounded and the infinite unwinding of recursion, respectively. The behavior described by the greatest fixed–point operator in LTμ thus corresponds to recursion in CCS. In the light of this discussion, LPC extends CCS by operators for *disjunction*, *conjunction*, and *minimal fixed–points*, as well as the basic processes *true* and *false*, and thereby allows for the encoding of both LTμ formulas and CCS processes in LPC (cf. Sec. 2).

The semantics of LPC is based on the testing approach of De Nicola and Hennessy [12]. The hallmarks of this theory are on the one hand the use of transitions to model both processes and tests and on the other hand the differentiation of processes on the basis of their responses to tests. Accordingly, we equip LPC terms with a transition relation defining the single–step transitions that specifications may engage in. We also introduce a novel *unimplementability predicate* on terms whose role is to identify inconsistent specifications, such as *false*, that cannot be implemented. Both the transition relation and the unimplementability predicate are defined via structural operational rules, i.e., in a syntax–driven fashion. We then carry over the definitions of must–testing in [12] to our setting and show that the resulting behavioral preorder (i) conservatively extends the traditional must–preorder between CCS specifications, (ii) is compositional for all operators in LPC, and (iii) naturally encodes the standard satisfaction relation between CCS processes and LTμ formulas (cf. Sec. 3). Thus, our framework may be seen to unify refinement–based and logic–based approaches to system specification, while facilitating component–based reasoning. Technically, this expressiveness follows from the mathematically coherent inclusion of process and logical operators in LPC that is enabled by our treatment of unimplementability (cf. Sec. 4). Practically, the theory allows system modelers to freely intermix operational and declarative subspecifications using both system operators (e.g. parallel composition) and logical constructors (e.g. conjunction). This gives engineers powerful tools to model system components at different levels of abstraction and to impose declarative constraints on the execution behavior of components (cf. Sec. 5).

2 A Logical Process Calculus

This section formally introduces our logical process calculus, LPC. We present its syntax, define its semantics via operational rules and a novel unimplementability predicate, and equip it with a refinement preorder on processes, which is adapted from De Nicola and Hennessy [12].

Syntax of LPC. The syntax of LPC extends Milner's CCS [26] with *disjunction*, *conjunction*, and *least fixed–point* operators. It also includes a process constant for the universal process *true*, while *false* will be a derived process term in our calculus. Formally, let Λ be a countable set of *actions*, or ports, not including the distinguished unobservable, *internal* action τ. With every $a \in \Lambda$ we associate a *complementary action* $\overline{a}$. We define $\overline{\Lambda} := \{\overline{a} \,|\, a \in \Lambda\}$ and take $\mathcal{A}$ to denote the set $\Lambda \cup \overline{\Lambda}$. Complementation is lifted to $\mathcal{A}$ by defining $\overline{\overline{a}} := a$. As in CCS, an action a communicates with its complement $\overline{a}$ to produce the internal action τ. We let $a, b, \ldots$ range over $\mathcal{A}$ and $\alpha, \beta, \ldots$ over $\mathcal{A}_\tau := \mathcal{A} \cup \{\tau\}$. The syntax of LPC is then defined as follows:

$$P \;::=\; \mathbf{0} \mid \mathsf{tt} \mid x \mid w \mid \alpha.P \mid P+P \mid P \vee P \mid P|P \mid P \wedge P \mid P \setminus L \mid P[f] \mid \mu x.P \mid \mu_k x.P \mid \nu x.P$$

where $k \in \mathbb{N}$, x is a *variable* taken from some nonempty set $\mathcal{V}$ of variables, w is an infinite word over $\mathcal{A}$ whose inclusion will be discussed in the next section, set $L \subseteq \mathcal{A}$ is a *restriction set*, and $f : \mathcal{A}_\tau \to \mathcal{A}_\tau$ is a *finite relabeling*. A finite relabeling satisfies the properties $f(\tau) = \tau$, $f(\overline{a}) = \overline{f(a)}$, and $|\{\alpha \,|\, f(\alpha) \neq \alpha\}| < \infty$. We define $\overline{L} := \{\overline{a} \,|\, a \in L\}$ and use the standard definitions for *free* and *bound* variables, *open* and *closed* terms, *guardedness*, and *contexts*. We require for fixed–point terms $\mu x.P$, $\mu_k x.P$, and $\nu x.P$ that x is guarded in P. Intuitively, $\mu x.P$ stands for finite *unbounded* unwindings of P, while $\mu_k x.P$ encodes finite unwindings of P *bounded* by k. A term is called *alternation–free* if every variable bound by a least (greatest) fixed–point $\mu x.P$ ($\nu x.P$) does not occur free in a subterm $\nu y.Q$ ($\mu y.Q$) of P. We refer to closed, guarded, and alternation–free [4] terms as *processes*, with the set of all processes written as $\mathcal{P}$. Finally, we denote syntactic equality by $\equiv$.

[4] The restriction to alternation–free processes is made for continuity reasons that are elaborated on later. Note that alternation–free processes still allow one to express fairness constraints, as will be demonstrated in Sec. 5.

While it is obvious that LPC subsumes all CCS processes, it is not immediately clear that it also encodes all Alternation–Free Linear–Time μ–Calculus ($\mathsf{LT}\mu$) formulas [5].[5] The syntax of $\mathsf{LT}\mu$ formulas is defined as follows:

$$\Phi \quad ::= \quad \mathbf{0} \;\mid\; \mathsf{tt} \;\mid\; \mathsf{ff} \;\mid\; x \;\mid\; \langle a \rangle \Phi \;\mid\; \Phi \vee \Phi \;\mid\; \Phi \wedge \Phi \;\mid\; \mu x.\Phi \;\mid\; \nu x.\Phi$$

In our setting, $\mathsf{LT}\mu$ formulas will be interpreted over infinite action sequences and also finite ones leading to deadlock. This is why the 'deadlock formula' $\mathbf{0}$ is included in $\mathsf{LT}\mu$. In LPC, ff corresponds to the term $\mu x.\tau.x$, as will become clear from our semantics definition below, and the *next* operator '$\langle a \rangle$', for $a \in \mathcal{A}$, corresponds to the prefix operator '$a.$'.

Semantics of LPC. The *operational semantics* of an LPC process P as a labeled transition system $\langle \mathcal{P}, \mathcal{A}_\tau, \longrightarrow, \#, P \rangle$, where $\mathcal{P}$ is the set of states, $\mathcal{A}_\tau$ the alphabet, $\longrightarrow \subseteq \mathcal{P} \times \mathcal{A}_\tau \times \mathcal{P}$ the transition relation, $\# \subseteq \mathcal{P}$ our *unimplementability predicate* that is discussed below, and P the start state.

The transition relation is defined by the *structural operational rules* displayed in Table 1. For convenience, we write $P \stackrel{\alpha}{\longrightarrow} P'$ instead of $\langle P, \alpha, P' \rangle \in \longrightarrow$. Note that, for the CCS operators, the semantics is exactly as in [26]. As for the other constructs, tt can nondeterministically engage in any action transition, or decide to deadlock (cf. Rules (True1) and (True2)). Process $\alpha.P$ may engage in action α and then behave like P (cf. Rule (Act1)), and similarly the process described by the infinite word aw may engage in its initial action a and then behave like w (cf. Rule (Act2)). The reason for including process w is to enable the modeling of arbitrary system environments within our calculus, including those exhibiting irregular behavior. The summation operator $+$ denotes *nondeterministic external choice* such that $P + Q$ may behave like P or Q, depending on which communication initially offered by P and Q is accepted by the environment (cf. Rules (Sum1) and (Sum2)). Analogously, $\vee$ encodes *disjunction* or *nondeterministic internal choice*, i.e., process $P \vee Q$ determines internally, without consulting its environment, whether to execute P or Q (cf. Rules (Dis1) and (Dis2)). Process $P|Q$ stands for the *asynchronous parallel composition* of processes P and Q according to an interleaving semantics with synchronized communication on complementary actions, resulting in the internal action τ (cf. Rules (Par1)–(Par3)). Similarly, $P \wedge Q$ encodes the *conjunction* or *synchronous parallel composition* of P and Q, with synchronization on all visible actions and interleaving on τ (cf. Rules (Con1)–(Con3)). The *restriction operator* $\setminus L$ prohibits the execution of actions in $L \cup \overline{L}$ and, thus, permits the scoping of actions. Process $P[f]$ behaves exactly as P where actions are renamed according to the *relabeling* f. The remaining rules define the semantics of our least and greatest fixed–point operators. The *minimal fixed–point* process $\mu x.P$ first guesses some number $k \in \mathbb{N}$ that determines how

[5] $\mathsf{LT}\mu$ is more expressive than linear–time temporal logic, so the limitation to alternation–free formulas does not impose undue expressiveness restrictions.

Table 1
Operational semantics

$$\text{True1}\ \frac{-}{\mathtt{tt} \xrightarrow{\tau} a.\mathtt{tt}}\ a \in \mathcal{A} \qquad \text{True2}\ \frac{-}{\mathtt{tt} \xrightarrow{\tau} \mathbf{0}}$$

$$\text{Act1}\ \frac{-}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{Act2}\ \frac{-}{aw \xrightarrow{a} w}$$

$$\text{Sum1}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{Sum2}\ \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\text{Dis1}\ \frac{-}{P \vee Q \xrightarrow{\tau} P} \qquad \text{Dis2}\ \frac{-}{P \vee Q \xrightarrow{\tau} Q}$$

$$\text{Par1}\ \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \text{Par2}\ \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$\text{Con1}\ \frac{P \xrightarrow{\tau} P'}{P \wedge Q \xrightarrow{\tau} P' \wedge Q} \qquad \text{Con2}\ \frac{Q \xrightarrow{\tau} Q'}{P \wedge Q \xrightarrow{\tau} P \wedge Q'}$$

$$\text{Par3}\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \text{Con3}\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \wedge Q \xrightarrow{a} P' \wedge Q'}$$

$$\text{Res}\ \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha \notin L \cup \overline{L} \qquad \text{Rel}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$\text{Mu1}\ \frac{-}{\mu x.P \xrightarrow{\tau} \mu_k x.P}\ k \in \mathbb{N} \qquad \text{Mu2}\ \frac{P[\mu_{k-1}x.P/x] \xrightarrow{\alpha} P'}{\mu_k x.P \xrightarrow{\alpha} P'}\ k > 0$$

$$\text{Nu}\ \frac{P[\nu x.P/x] \xrightarrow{\alpha} P'}{\nu x.P \xrightarrow{\alpha} P'}$$

often P might be unwound, as encoded by the process $\mu_k x.P$ (cf. Rules (Mu1) and (Mu2)). Here, $P[Q/x]$ stands for the process P with all of its free occurrences of variable x substituted by Q. This account of μ may be seen as embodying a form of *continuity*: μ is interpreted in terms of its finite unwindings. Because of continuity problems associated with alternating least and greatest fixed points, which are well–documented in the literature [33], we only consider alternation–free process expressions in this paper. The *maximal fixed–point* process $\nu x.P$ may unwind its loop indefinitely, as is the case for recursion in CCS (cf. Rule (Nu)). Note that the purely *divergent process* Ω, employed in some process algebras [17] for describing infinite internal computation and already expressible in CCS [26], can be derived in LPC as $\nu x.\tau.x$.

Table 2
Unimplementability predicate $\#$

(i) $\mu_0 x.P\,\#$

(ii) $(P \longrightarrow$ and $P \wedge Q \not\longrightarrow)$ implies $P \wedge Q\,\#$

(iii) $(Q \longrightarrow$ and $P \wedge Q \not\longrightarrow)$ implies $P \wedge Q\,\#$

(iv) $P\,\#$ implies

- $\alpha.P\,\#$ • $P[f]\,\#$ • $P \setminus L\,\#$
- $P + Q\,\#$ • $Q + P\,\#$
- $P \wedge Q\,\#$ • $Q \wedge P\,\#$
- $P|Q\,\#$ • $Q|P\,\#$
- $\nu x.P\,\#$ • $\mu x.P\,\#$ • $\mu_k x.P\,\#$, for all $k \in \mathbb{N}$

(v) $P\,\#$ and $Q\,\#$ implies $P \vee Q\,\#$

(vi) $P[\mu_{k-1}x.P/x]\,\#$ implies $\mu_k x.P\,\#$, for all $k > 0$

(vii) $(\forall k \in \mathbb{N}.\ \mu_k x.P\,\#)$ implies $\mu x.P\,\#$

Temporal logics, including $\mathsf{LT}\mu$, are capable of specifying *inconsistencies* or contradictions, i.e., behaviors equivalent to *false*. From an operational point of view, a process describing an inconsistency is not implementable, and thus runs of processes passing through unimplementable states should be ignored. Due to logical disjunction, however, a process that can engage in such runs is not necessarily unimplementable itself. Note the difference between unimplementability for logical disjunction $P \vee Q$ and nondeterministic choice $P + Q$: The latter process $P + Q$ denotes a completely operational process that is implementable if both P and Q are implementable. In contrast, $P \vee Q$ can be implemented if either P or Q can.

This intuition is reflected in the definition of our *unimplementability predicate*, given in Table 2, where we write $P\,\#$ for $P \in \#$ and where $P \longrightarrow$ stands for $\exists P' \in \mathcal{P}.\, \exists \alpha \in \mathcal{A}_\tau\ P \stackrel{a}{\longrightarrow} P'$. In particular, a contradiction is present within a conjunction $P \wedge Q$, if the conjunction process cannot engage in any transition, although one of its argument processes can (cf. Rules (ii) and (iii)). As an example, consider process $a.\mathbf{0} \wedge b.\mathbf{0}$, for $a \not\equiv b$. Further, the first part of Rule (iv) states that the unimplementability of P propagates backwards through prefixing. Note that the operational semantics for LPC distinguishes between inconsistent processes, which are unimplementable, and deadlocked processes, which are implementable. For example, both processes $(a.\mathsf{0}|b.\mathsf{0}) \setminus \{a, b\}$ and $a.\mathsf{0} \wedge b.\mathsf{0}$ cannot engage in any transition. However, $(a.\mathsf{0} \wedge b.\mathsf{0})\,\#$ while $\neg(((a.\mathsf{0}|b.\mathsf{0}) \setminus \{a, b\})\,\#)$, as desired. All other rules are straightforward, except for least fixed–point processes, such as the process $\mu_0 x.P$ that cannot unwind its body P further and is thus considered to be unimplementable (cf. Rule (i)). Together with Rules (vi) and (vii), this implies that the process $\mu x.\tau.x$, which can engage in finite but unbounded numbers of τ's, is actually unimplementable. Indeed, we will identify this process with *false* and abbreviate it by ff. It is this definition that will allow one to distinguish the processes ff and $\mathsf{0}$.

The semantics for LPC does not only extend the standard CCS semantics but is also compatible with the semantics of $\mathsf{LT}\mu$ formulas; see Thm. 3.5. This theorem, however, is not straightforward, and its proof requires us to build a rich semantic theory for LPC. Before doing so we first introduce some notation. A *potential path* π of process P is a sequence of transitions $(P_i \xrightarrow{\alpha_i} P_{i+1})_{0 \leq i < k}$, for some $k \in \mathbb{N} \cup \{\omega\}$, such that $P_0 \equiv P$. If $\neg(P_i\,\#)$, for all $0 \leq i < k$ and for $i = k$ if $k \in \mathbb{N}$, then π is called an *implementable path*, or simply *path*. We use $|\pi|$ to refer to k, the length of π. If $|\pi| = \omega$, we say that π is *infinite*; otherwise, π is *finite*. Moreover, π is called *maximal* if $|\pi| < \omega$ and $P_{|\pi|} \not\longrightarrow$. The *trace* $\mathsf{trace}(\pi)$ of π is defined as the word $w := (\alpha_i)_{I_\pi} \in \mathcal{A}^\infty := \mathcal{A}^* \cup \mathcal{A}^\omega$, where $I_\pi := \{0 \leq i < |\pi| \,|\, \alpha_i \not\equiv \tau\}$. In the case of $I_\pi = \emptyset$, we let ϵ stand for $w = ()$. Moreover, if π is finite, we also write $P \overset{w}{\Longrightarrow} P_{|\pi|}$ for π. We denote the sets of all finite, maximal, and infinite paths of P by $\Pi_{\mathsf{fin}}(P)$, $\Pi_{\mathsf{max}}(P)$, and $\Pi_\omega(P)$, respectively. We may also introduce according languages for P:

$$
\begin{array}{llll}
\mathcal{L}_{\mathsf{fin}}(P) & := \{\mathsf{trace}(\pi) \,|\, \pi \in \Pi_{\mathsf{fin}}(P)\} & \subseteq \mathcal{A}^* & \textit{finite–trace language of } P \\
\mathcal{L}_{\mathsf{max}}(P) & := \{\mathsf{trace}(\pi) \,|\, \pi \in \Pi_{\mathsf{max}}(P)\} & \subseteq \mathcal{A}^* & \textit{maximal–trace language of } P \\
\mathcal{L}_{\omega}(P) & := \{\mathsf{trace}(\pi) \,|\, \pi \in \Pi_{\omega}(P)\} & \subseteq \mathcal{A}^\infty & \textit{infinite–trace language of } P
\end{array}
$$

The semantic theory to be developed for LPC relies on the notion of *divergence*, i.e., a system's ability to engage in an infinite internal computation. In this paper, we employ the traditional notion of divergence as used by De Nicola and Hennessy [12]; more sophisticated definitions may be found elsewhere in the literature [6,27,29]. Process P is *divergent*, in signs $P \Uparrow$, if $\epsilon \in \mathcal{L}_\omega(P)$. For example, process $\Omega := \nu x.\tau.x$ is divergent. A process P is called w–divergent for some $w \in \mathcal{A}^\infty$, in signs $P \Uparrow w$, if $\exists P' \in \mathcal{P}\, \exists v <_{\mathsf{fin}} w.\ P \overset{v}{\Longrightarrow} P'$ and $P' \Uparrow$. Here, $<_{\mathsf{fin}}$ stands for the finite prefix ordering on words. We further write $\mathcal{L}_{\mathsf{div}}(P)$ for the *divergent–trace language* of P, i.e., $\mathcal{L}_{\mathsf{div}}(P) := \{w \in \mathcal{A}^\infty \,|\, P \Uparrow w\}$. Finally, P is called *convergent* or w*–convergent*, in symbols $P \Downarrow$ and $P \Downarrow w$, if $\neg(P \Uparrow)$ and $\neg(P \Uparrow w)$, respectively.

Refinement in LPC. We now turn our attention to a behavioral theory of LPC, which defines a behavioral preorder $\sqsubseteq$ on processes such that $P \sqsubseteq Q$, i.e., Q refines P, if Q is "more defined" than P. The preorder is an adaptation of De Nicola and Hennessy's *must–preorder* [12], which was developed within an elegant *testing theory* and distinguishes processes on the basis of the *tests* they are necessarily able to pass. In this context, tests are processes equipped with a special action $\surd$, which are employed to witness the interactions a process may have with its environment. In order to determine whether a process passes a test, one has to examine the maximal and infinite *computations* that result when the test runs in lock–step with the process under consideration.

Formally, a *test* is a process that might use the distinguished success action $\surd \notin \mathcal{A}_\tau$. The set of all tests is denoted by $\mathcal{T}$. A *maximal (infinite) computation* π of process P and test T is a maximal (infinite) path π of $(P|T) \setminus \mathcal{A}$, i.e., $\pi = ((P_i|T_i) \setminus \mathcal{A} \stackrel{\tau}{\longrightarrow} (P_{i+1}|T_{i+1}) \setminus \mathcal{A})_{0 \le i < |\pi|}$. Recall that paths only go along implementable states (including the final state in a maximal computation). Computation π is *successful* if $T_i \stackrel{\surd}{\longrightarrow}$ for some $0 \le i < |\pi|$; otherwise, it is *unsuccessful*. Finally, process P is said to *must–satisfy* test T, in symbols $P \,\mathsf{must}\, T$, if every maximal and infinite computation of P and T is successful.

Definition 2.1 [Must–preorder] For $P, Q \in \mathcal{P}$ we let $P \sqsubseteq Q$ if, for all $T \in \mathcal{T}$, $P \,\mathsf{must}\, T$ implies $Q \,\mathsf{must}\, T$.

It is easy to see that $\sqsubseteq$ is a preorder, i.e., that it is reflexive and transitive. Note that this preorder can be extended to open terms by the usual means of closed substitution [26]. Moreover, $\sqsubseteq$ satisfies the following basic algebraic laws, where $\approx$ stands for the kernel $\sqsubseteq \cap (\sqsubseteq)^{-1}$ of $\sqsubseteq$.

Proposition 2.2 *Let $P, Q, R \in \mathcal{P}$. Then, the following holds:*

$$\begin{array}{cccc} P|\mathbf{0} \approx P & P|\Omega \approx \Omega & P \wedge \mathsf{tt} \approx P & P \wedge \mathsf{ff} \approx \mathsf{ff} \\ P + \mathbf{0} \approx P & P + \Omega \approx \Omega & P \vee \mathsf{tt} \approx \mathsf{tt} & P \vee \mathsf{ff} \approx P \end{array}$$

Further, $P \wedge P \approx P$, $P \vee P \approx P$, and $P \vee Q \sqsubseteq P$. All binary operators are, of course, also commutative and associative.

It is also easy to see that the divergent process Ω does not must–satisfy any tests, except the trivial ones, such as $\surd.\mathbf{0}$. Hence, it is the smallest process with respect to $\sqsubseteq$. Conversely, process ff must–satisfies every test, since it does not possess any computation due to $\mathsf{ff}\,\#$. Consequently, ff is the largest process with respect to $\sqsubseteq$. Also tt is a distinguished process in our setting; it is the smallest *convergent* process with respect to $\sqsubseteq$. Thus, we have $\Omega \sqsubseteq \mathsf{tt} \sqsubseteq \mathbf{0} \sqsubseteq \mathsf{ff}$; it is easy to verify that this ordering is actually strict.[6]

[6] This ordering is the reverse of the more usual Boolean ordering, which considers ff to be lower than tt, and arises since must refinement implies *reverse* language containment.

3 Properties of the Must–Preorder

In this section we investigate the utility of our calculus for the heterogeneous specification of reactive systems. We show that our must–preorder is a conservative extension of the one of De Nicola and Hennessy, provide its characterization in terms of traces and initial action sets, investigate its close relation to $\mathsf{LT}\mu$ satisfaction, and finally establish its compositionality properties. Due to space constraints we can only include some proof sketches here.

Extension of De Nicola and Hennessy's Must–Preorder. It is easy to see that our must–preorder $\sqsubseteq\!\!\!\!\sim$ is a conservative extension of the must–preorder $\sqsubseteq\!\!\!\!\sim_{\mathrm{DH}}$ of De Nicola and Hennessy, defined on CCS processes [12]. Indeed, their and our definitions of the testing framework coincide on CCS processes and CCS tests, which leads to the following conservativity theorem.

Theorem 3.1 *Let P, Q be CCS processes. Then, $P \sqsubseteq\!\!\!\!\sim Q$ if and only if $P \sqsubseteq\!\!\!\!\sim_{\mathrm{DH}} Q$.*

Characterization. We now present a characterization of our must–preorder which will be used for obtaining some of our main results. The characterization closely follows the lines of a similar characterization of De Nicola and Hennessy's must–preorder [12]. It uses the notation $\mathcal{I}(P)$ for the set $\{a \in \mathcal{A} \mid P \stackrel{\tau}{\longrightarrow}^* \stackrel{a}{\longrightarrow}\}$ of visible initial actions of P.

Theorem 3.2 *Let P and Q be processes. Then $P \sqsubseteq\!\!\!\!\sim Q$ if and only if for all $w \in \mathcal{A}^\infty$ such that $P \Downarrow w$:*

(i) $Q \Downarrow w$

(ii) $|w| < \omega$: $\forall Q'.\, Q \stackrel{w}{\Longrightarrow} Q'$ *implies* $\exists P'.\, P \stackrel{w}{\Longrightarrow} P'$ *and* $\mathcal{I}(P') \subseteq \mathcal{I}(Q')$
$|w| = \omega$: $w \in \mathcal{L}_\omega(Q)$ *implies* $w \in \mathcal{L}_\omega(P)$

Observe that this characterization is also sensitive to infinite traces and not only finite ones (cf. Cond. (2)). This is superficially similar to the *improved failures model* of [7]; the difference is that infinite traces in [7] convey divergence information, while they convey convergence information in the above characterization. The proof of the above theorem, which proceeds along the lines of the proof of a corresponding theorem that can be found in [9], is partly non–standard in that it relies on the following distinguished tests, where $k \in \mathbb{N}$, $w = (a_i)_{0 \leq i < k} \in \mathcal{A}^*$, $v \in \mathcal{A}^\omega$, and $a \in \mathcal{A}$.

(i) $T_w^{\Downarrow} := a_0.a_1.\cdots.a_{k-1}.\mathbf{0} \,|\, \tau.\surd.\mathbf{0}$

(ii) $T_w^{\mathsf{fin}} := a_0.(a_1.\cdots.(a_{k-1}.\mathbf{0} + \tau.\surd.\mathbf{0})\cdots) + \tau.\surd.\mathbf{0}) + \tau.\surd.\mathbf{0}$

(iii) $T_{w,a}^{\mathsf{max}} := a_0.(a_1.\cdots.(a_{k-1}.a.\surd.\mathbf{0} + \tau.\surd.\mathbf{0})\cdots) + \tau.\surd.\mathbf{0}) + \tau.\surd.\mathbf{0}$

(iv) $T_v^{\omega} := v \,|\, \tau.\surd.\mathbf{0}$

The intuitions behind defining these tests are as follows.

Lemma 3.3 *Let P be an arbitrary LPC process and*

(i) *Let $w \in \mathcal{A}^*$. Then, $P \Downarrow w$ iff P must $T_w^{\Downarrow}$.*
(ii) *Let $w \in \mathcal{A}^*$ such that $P \Downarrow w$. Then, $w \notin \mathcal{L}_{\mathsf{fin}}(P)$ iff P must T_w^{fin}.*
(iii) *Let $w \in \mathcal{A}^*$ such that $P \Downarrow w$. Then, $w \notin \mathcal{L}_{\mathsf{max}}(P)$ iff $\exists a \in \mathcal{A}$. P must $T_{w,a}^{\mathsf{max}}$.*
(iv) *Let $v \in \mathcal{A}^\omega$ such that $P \Downarrow v$. Then, $v \notin \mathcal{L}_\omega(P)$ iff P must T_v^ω.*

The proof of this lemma is not too difficult but tedious; it follows our definition of must–passing tests and is similar to a corresponding proof in [10]. Note that the first property can also be carried over to infinite words, due to our 'approximative' definition of divergence.

Extension of LTμ Satisfaction. To prove that our must–preorder is also an extension of LTμ satisfaction we first recall the standard semantics of LTμ. An LTμ formula is interpreted as the set of those finite and infinite sequences over $\mathcal{A}$ that validate the formula. Formally, the semantics $[\![\Phi]\!]^{\mathcal{E}}$ of a possibly open LTμ term Φ is defined relative to an environment $\mathcal{E}$ mapping variables to subsets of $\mathcal{A}^\infty$. Note that our variant of the linear–time μ–calculus [5] can be used to reason about deadlock traces as well, due to our inclusion of the atomic proposition $\mathbf{0}$; this is why we also consider finite traces.

$$
\begin{array}{rclrclrcl}
[\![\mathsf{tt}]\!]^{\mathcal{E}} &:=& \mathcal{A}^\infty & [\![\mathsf{ff}]\!]^{\mathcal{E}} &:=& \emptyset & [\![x]\!]^{\mathcal{E}} &:=& \mathcal{E}(x) \\
[\![\langle a\rangle\Phi]\!]^{\mathcal{E}} &:=& \multicolumn{4}{l}{\{aw \mid w \in [\![\Phi]\!]^{\mathcal{E}}\}} & [\![\mathbf{0}]\!]^{\mathcal{E}} &:=& \{\epsilon\} \\
[\![\mu x.\Phi]\!]^{\mathcal{E}} &:=& \multicolumn{4}{l}{\bigcap\{T \subseteq \mathcal{A}^\infty \mid [\![\Phi]\!]^{\mathcal{E}[x\mapsto T]} \subseteq T\}} & [\![\Phi_1 \wedge \Phi_2]\!]^{\mathcal{E}} &:=& [\![\Phi_1]\!]^{\mathcal{E}} \cap [\![\Phi_2]\!]^{\mathcal{E}} \\
[\![\nu x.\Phi]\!]^{\mathcal{E}} &:=& \multicolumn{4}{l}{\bigcup\{T \subseteq \mathcal{A}^\infty \mid T \subseteq [\![\Phi]\!]^{\mathcal{E}[x\mapsto T]}\}} & [\![\Phi_1 \vee \Phi_2]\!]^{\mathcal{E}} &:=& [\![\Phi_1]\!]^{\mathcal{E}} \cup [\![\Phi_2]\!]^{\mathcal{E}}
\end{array}
$$

In case Φ is a formula, i.e., Φ is a closed LTμ term, it is easy to see that the environment $\mathcal{E}$ is irrelevant. We say that a CCS process P satisfies Φ, in signs $P \models \Phi$, if all traces of P are included in the traces of $[\![\Phi]\!]$. Formally, $P \models \Phi$ if (i) $\mathcal{L}_{\mathsf{div}}(P) \subseteq \mathcal{L}_{\mathsf{div}}(\Phi)$, (ii) $\mathcal{L}_{\mathsf{max}}(P) \subseteq [\![\Phi]\!]$, and (iii) $\mathcal{L}_\omega(P) \subseteq [\![\Phi]\!]$.

Further, LTμ formulas, when considered as a sublanguage of LPC, possess two important properties. First, all formulas Φ are convergent, i.e., $\mathcal{L}_{\mathsf{div}}(\Phi) = \emptyset$. This is because the internal prefix operator 'τ.' is not available in LTμ. In addition, the atomic propositions tt, ff, and $\mathbf{0}$ do not give rise to divergence. As a consequence, Cond. (i) in the definition of $P \models \Phi$ above can be simplified to $\mathcal{L}_{\mathsf{div}}(P) = \emptyset$. In particular, formula tt is satisfied by convergent processes only, whence $P \models \mathsf{tt}$ if and only if $\mathcal{L}_{\mathsf{div}}(P) = \emptyset$. Second, every LT$\mu$ formula Φ is purely nondeterministic in the sense that all choices are internal:

$$\forall \Phi', \Phi''\ \forall \alpha, \beta.\ \Phi \xrightarrow{\alpha} \Phi',\ \Phi \xrightarrow{\beta} \Phi'',\ \Phi' \not\equiv \Phi'' \text{ implies } \alpha \equiv \beta \equiv \tau\ .$$

This is due to the fact that disjunction is modeled as internal choice in LPC.

Proposition 3.4 *Let Φ be an LTμ formula and P a CCS process. Then, $\Phi \sqsubseteq\!\!\!\sim P$ if and only if (i) $\mathcal{L}_{\mathsf{div}}(P) = \emptyset$, (ii) $\mathcal{L}_{\mathsf{max}}(P) \subseteq \mathcal{L}_{\mathsf{max}}(\Phi)$, (iii) $\mathcal{L}_\omega(P) \subseteq \mathcal{L}_\omega(\Phi)$.*

The proof of this proposition relies on our characterization theorem for $\underset{\sim}{\sqsubset}$ (cf. Thm. 3.2) and uses the two properties of formulas mentioned above. The proposition is the key for establishing the next theorem.

Theorem 3.5 *Let P be a* CCS *process and Φ an* LTμ *formula. Then, $P \models \Phi$ if and only if $\Phi \underset{\sim}{\sqsubset} P$.*

Due to Prop. 3.4 and the definition of $\models$, it is sufficient to prove that $[\![\Phi]\!] = \mathcal{L}_{\mathsf{max}}(\Phi) \cup \mathcal{L}_{\omega}(\Phi)$. This can be done along the structure of $\mathsf{LT}\mu$ formulas, but requires the appropriate extension of the definition of languages to open terms.

The above theorem also establishes that the LPC operator '$\wedge$' is indeed a logical conjunction operator when restricted to operands in $\mathsf{LT}\mu$. Formally, $\Phi_1 \wedge \Phi_2 \underset{\sim}{\sqsubset} P$ if and only if $\Phi_1 \underset{\sim}{\sqsubset} P$ and $\Phi_2 \underset{\sim}{\sqsubset} P$, for all $\mathsf{LT}\mu$ formulas Φ_1, Φ_2 and CCS processes P; this statement also holds for arbitrary LPC processes P.

Compositionality. One virtue of process algebras is that they allow one to reason compositionally about processes. Our logical process calculus LPC is no exception. Indeed our must–preorder is compositional for all operators, except for the choice operators $+$ and $\vee$. This defect manifests itself also in De Nicola and Hennessy's must–preorder. The largest precongruence $\sqsubseteq$ contained in $\underset{\sim}{\sqsubset}$ can be obtained in the standard way [12].

Definition 3.6 [Must–precongruence] For $P, Q \in \mathcal{P}$ we write $P \sqsubseteq Q$ if (i) $P \underset{\sim}{\sqsubset} Q$ and (ii) $Q \stackrel{\tau}{\longrightarrow}$ implies $P \stackrel{\tau}{\longrightarrow}$.

Theorem 3.7 *The preorder $\sqsubseteq$ is a precongruence, i.e., for all processes P, Q such that $P \sqsubseteq Q$, we have:*

- $\alpha.P \sqsubseteq \alpha.Q$ *for all* $\alpha \in \mathcal{A}$
- $P + R \sqsubseteq Q + R$ *for all* $R \in \mathcal{P}$
- $P \vee R \sqsubseteq Q \vee R$ *for all* $R \in \mathcal{P}$
- $P|R \sqsubseteq Q|R$ *for all* $R \in \mathcal{P}$
- $P \wedge R \sqsubseteq Q \wedge R$ *for all* $R \in \mathcal{P}$
- $P \setminus L \sqsubseteq Q \setminus L$ *for all restriction sets* L
- $P[f] \sqsubseteq Q[f]$ *for all relabelings* f
- $\mu_k x.P \sqsubseteq \mu_k x.Q$ *for all* $x \in \mathcal{V}$, $k \in \mathbb{N}$
- $\mu x.P \sqsubseteq \mu x.Q$ *for all* $x \in \mathcal{V}$
- $\nu x.P \sqsubseteq \nu x.Q$ *for all* $x \in \mathcal{V}$

Moreover, $\sqsubseteq$ is the largest precongruence contained in $\underset{\sim}{\sqsubset}$.

Compositionality can be checked straightforwardly for most operators, except for the largest fixed–point operator, by referring to Thm. 3.2. Regarding asynchronous parallel composition, the compositionality of $\sqsubseteq$ follows directly from the fact that $P|Q \,\mathsf{must}\, T$ if and only if $P \,\mathsf{must}\, Q|T$, for all $P, Q \in \mathcal{P}$ and $T \in \mathcal{T}$; this is essentially the associativity property of $|$. In case of the largest fixed–point operator, one needs to reason indirectly via a *denotational* characterization of our operational semantics in terms of a suitably modified form of *acceptance trees* [12]; unfortunately, the presentation of this denotational characterization here is made impossible by space constraints. The proof of the 'largest' statement in Thm. 3.7 is standard [12].

4 Discussion and Related Work

This section contrasts LPC to related work and discusses the fundamental differences of the setting presented here to our previous approach [10].

Most early related work couples operational and declarative approaches to system specification loosely and does not allow for mixed specifications. This includes the large amount of work on relating behavioral equivalences or preorders to temporal logics in one of the following ways: (i) establishing that one system refines another if and only if both satisfy the same temporal formulas [13,18,26,32]; (ii) translating finite–state labeled transition systems into temporal formulas [31]; or (iii) encoding subclasses of temporal formulas as behavioral relations via the idea of implicit specifications [24]. Other work, in the field of compositional model checking [8,15,21], aimed at supporting a modular approach for reasoning about temporal–logic specifications. Several researchers have also considered the inclusion of different fixed–point operators in behavioral theories of processes in order to model fairness and unbounded but finite delay [16,19]. One may also find a process algebra with an element similar to our process ff in [2]. Diverting from these approaches, advanced frameworks for genuine heterogeneous specifications have been developed as well, which can be distinguished according to whether they employ logic/algebraic or automata–theoretic techniques.

Logic/algebraic approaches. This category includes the seminal work of Abadi and Lamport, who have developed ideas for heterogeneous specifications for shared–memory systems [1]. Their technical setting is the logical framework of TLA [23], in which processes and temporal formulas are indistinguishable and logical implication serves as the refinement relation. The difference to our setting is that TLA refinement is insensitive to deadlock and divergence. While this might not be a problem for shared–memory systems, it is not suitable for reasoning about distributed systems, at which our calculus LPC aims. Graf and Sifakis follow a similar line of development in [14]. There, a logic is developed that includes constructs for actions and nondeterministic choice, and a logical encoding of operational behavior is given. In this logic, one establishes that a system satisfies a property by showing that the logical formula associated with the system implies the property.

In a different line of research, Valmari et al. have studied several congruences preserving "next–time–less" linear–time temporal logic [28], which may also handle deadlock and livelock [20,29,34]. A good overview by Puhakka and Valmari on the matters of liveness and fairness in process algebra can be found in [30]. This paper also observes that, during system refinement, fairness constraints are often only relevant for intermediate systems and are automatically implied when considering the larger system context. It then suggests a way to avoid constructing the usually infinite intermediate systems. Our work complements theirs in that LPC allows for embedding arbitrary LTL formulas in operational specifications, instead of a specific class of fairness constraints.

However, LPC does not avoid reasoning about infinite intermediate systems which can, in our opinion, be handled by employing clever data structures for implementing our must–preorder in verification tools, such as the *Concurrency Workbench NC* [11]. Finally, it should be noted that De Nicola and Hennessy's testing theory [12] has also been enriched with notions of fairness [6,27], in order to constrain infinite computations in labeled transition systems.

Automata–theoretic approaches. Regarding automata–theoretic techniques, the work of Kurshan is of direct relevance to this paper [22], who presented a theory of ω–word automata that includes notions of synchronous and asynchronous composition. However, Kurshan's underlying semantic model maps processes to their infinite traces, and the associated notion of refinement is (reverse) trace inclusion. In theories of concurrency, such as in ours in which deadlock is possible, maximal trace inclusion is not compositional [25].

The most closely related approach to the one presented here was introduced by the authors in [10]. Büchi automata were employed to uniformly encode mixed operational and declarative behavior, exploiting the well–known relation between Büchi automata and LTL [35]. We equipped this semantic framework with a notion of Büchi must–testing that extends De Nicola and Hennessy's must–testing preorder from labeled transition systems to Büchi automata. The intuition was to consider only those infinite traces as infinite computations that go through Büchi states infinitely often, and only to accept those infinite computations for which the considered Büchi test declares success infinitely often. The relation of our Büchi must–preorder to the LTL satisfaction relation, with the central result intended to be analogous to Thm. 3.5, was then established in a pure automata–theoretic fashion by suitably adapting the construction of [35]. However, our previous approach had several shortcomings that made it unsuitable as a semantic basis for a logical process calculus; these are discussed next.

Most importantly, our paper [10] contained a subtle technical mistake in the analogue of Lemma 3.3, which propagated through the paper's results. In a nutshell, the setup of Büchi testing did not allow us, as was intended, to ignore non–Büchi divergent traces, i.e., those infinite internal computations that go through Büchi states only finitely often. While most of the results in [10] could be repaired by explicitly observing non–Büchi divergence, the framework did no longer reflect the underlying intuition, and it made compositionality difficult to achieve for some operators, including parallel composition. Moreover, our identification of ff, or other inconsistent specifications, with non–Büchi divergence led to the invalidity of the desired law $P \vee \mathsf{ff} \approx P$. The present paper repairs this defect by associating ff with a process that cannot engage in any observable transition, nor in any divergence. In order to then distinguish ff from, say, $\mathbf{0}$ we introduced the unimplementability predicate. Similar difficulties arose when interpreting tt as Büchi–divergent process, which is why this paper distinguishes between tt and Ω, making tt the smallest *convergent* process in our must–preorder, while Ω still is the smallest process overall.

Indeed, the collection of these insights also allowed us to do away with Büchi automata as our semantic framework for heterogeneous system design altogether. Accordingly, LPC encodes the least and greatest fixed–points occurring in temporal logics via labeled transition systems, where the process–algebraic semantic rules for least fixed–points reflect the intuition that the recursion under consideration can only be unwound finitely often, while a recursion associated with a greatest fixed–point may be unwound infinitely often. Hence, in LPC all infinite traces are 'good', which means that the expressive power of Büchi automata to distinguish 'good' and 'bad' infinite traces is no longer needed. The result is a process calculus, LPC, in which classical process algebras and linear–time temporal logics can be uniformly integrated, as was envisioned in [10]. The example in the next section highlights the expressiveness of LPC[7] as well as its underlying practical motivation.

5 Example: Heterogeneous System Design

We illustrate by means of an example, the kind of refinement–based system design supported by LPC. The example advocates a heterogeneous style of system specification, combining process–algebraic and temporal–logic specifications, and thereby testifies to the utility of our calculus. It will be convenient to express temporal constraints by means of formulas in *Linear–time Temporal Logic* (LTL) [28] — a temporal logic that engineers often prefer over the linear–time μ–calculus [5]. We thus briefly show how LTL formulas can be encoded in $\mathsf{LT}\mu$ or, more precisely, in our new calculus LPC.

Encoding of LTL in LPC. Since we would like to describe action–based distributed systems and their deadlock behavior, the variant of LTL studied here includes the atomic propositions a, for $a \in \mathcal{A}$, and $\mathbf{0}$. Note that, in the context of temporal logics, $\mathcal{A}$ is always taken to be a finite set.

$$\Phi \quad ::= \quad \mathbf{0} \mid a \mid \mathsf{tt} \mid \mathsf{ff} \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \mathsf{X}\Phi \mid \hat{\mathsf{X}}\Phi \mid \Phi \mathsf{U} \Phi \mid \Phi \mathsf{V} \Phi$$

The temporal operators X, U, and V are intuitively interpreted as *next*, *until*, and *release* operators, respectively. Operator $\hat{\mathsf{X}}$ is the dual operator of X, namely a *next* operator that tolerates deadlocks; note that X is not self–dual in the presence of finite traces. An LTL formula Φ corresponds to the LPC process $\{\!\![\Phi]\!\!\}$, where the translation function $\{\!\![\cdot]\!\!\}$ is defined along the structure of Φ as follows and where x is some randomly chosen variable in $\mathcal{V}$.

$$\begin{array}{llll} \{\!\![\mathbf{0}]\!\!\} := \mathbf{0} & \{\!\![\mathsf{tt}]\!\!\} := \mathsf{tt} & \{\!\![\Phi_1 \vee \Phi_2]\!\!\} := \{\!\![\Phi_1]\!\!\} \vee \{\!\![\Phi_2]\!\!\} & \{\!\![\mathsf{X}\Phi]\!\!\} := \bigvee_{a \in \mathcal{A}} a.\{\!\![\Phi]\!\!\} \\ \{\!\![a]\!\!\} := a.\mathsf{tt} & \{\!\![\mathsf{ff}]\!\!\} := \mathsf{ff} & \{\!\![\Phi_1 \wedge \Phi_2]\!\!\} := \{\!\![\Phi_1]\!\!\} \wedge \{\!\![\Phi_2]\!\!\} & \{\!\![\hat{\mathsf{X}}\Phi]\!\!\} := \mathbf{0} \vee \bigvee_{a \in \mathcal{A}} a.\{\!\![\Phi]\!\!\} \end{array}$$

[7] It seems doubtful to us whether LPC can be encoded in standard CCS, which appears to be problematic regarding the conjunction and least–fixed–point operators.

$$\{\!\![\Phi_1 \mathsf{U} \Phi_2]\!\!\} := \mu x. \{\!\![\Phi_2]\!\!\} \ \vee \ (\{\!\![\Phi_1]\!\!\} \ \wedge \ \bigvee_{a \in \mathcal{A}} a.x)$$

$$\{\!\![\Phi_1 \mathsf{V} \Phi_2]\!\!\} := \nu x. \{\!\![\Phi_2]\!\!\} \ \wedge \ (\{\!\![\Phi_1]\!\!\} \ \vee \ \mathbf{0} \ \vee \ \bigvee_{a \in \mathcal{A}} a.x)$$

For convenience, we abbreviate formula $\mathsf{ff}\mathsf{V}\Phi$ by $\mathsf{G}\,\Phi$ ("*generally* Φ") and $\mathsf{tt}\mathsf{U}\Phi$ by $\mathsf{F}\,\Phi$ ("*eventually* Φ"), as usual. Moreover, we let $a \implies \Phi$ stand for the process $a.\Phi \vee \mathbf{0} \vee \bigvee_{b \not\equiv a} b.\mathsf{tt}$ which is valid if and only if, for all traces of the form aw, trace w satisfies Φ.

Example. Suppose an engineer designs a reliable bidirectional network link in a component–based fashion. One might think of this link as a composition of two reliable unidirectional links that are closely tight together. In particular, the failure of one unidirectional link should imply the failure of the other, which is a typical physical constraint of bidirectional links. The engineer might begin with a simple specification of an unreliable unidirectional link,

$$\mathtt{ULSpec} \ := \ \nu x.\overline{\mathtt{up}}.(x + \overline{\mathtt{fail}}.\nu y.\overline{\mathtt{down}}.(y \vee x)) \,,$$

which signals whether the link is up or down, or whether it just failed. In case of failure, the link tries to repair itself and, if and once it is successfully repaired, it returns to its initial state. However, a successful repair is not guaranteed, whence the process `ULSpec` may infinitely engage in the $\overline{\mathtt{down}}$–loop over variable y.

To obtain a specification `RLSpec` of a reliable unidirectional link, `ULSpec` is simply refined by adding a constraint imposing a "repair guarantee," $\mathtt{RG} := \mathsf{G}\,(\mathtt{fail} \implies \mathsf{F}\,\overline{\mathtt{up}})$, i.e., every broken link is eventually repaired and up. We then define $\mathtt{RLSpec} := \mathtt{ULSpec} \wedge \mathtt{RG}$, which does away with the $\overline{\mathtt{down}}$–loop in `ULSpec`. The desired bidirectional link might then be specified as follows:

$$\begin{aligned}\mathtt{BLSpec} := (\, & \mathtt{RLSpec}[\mathtt{up1}/\mathtt{up}, \mathtt{down1}/\mathtt{down}, \mathtt{sync}/\mathtt{fail}] \\ \mid\, & \mathtt{RLSpec}[\mathtt{up2}/\mathtt{up}, \mathtt{down2}/\mathtt{down}, \overline{\mathtt{sync}}/\mathtt{fail}] \\ )\, & \setminus \{\mathtt{sync}\} \,,\end{aligned}$$

where the synchronization on action `fail`, via the relabeling to action `sync`, ensures that the failure of one unidirectional link implies the failure of the other. Note that the constraints `RG` indirectly refer to action `sync`, which is restricted in `BLSpec`.

The engineer may now refine the heterogeneous LPC specification `BLSpec` into a pure CCS implementation. The idea is to fulfill the constraints `RG` by eliminating the $\overline{\mathtt{down}}$–loop in `ULSpec`, thus encoding that a repair can always be successfully carried out immediately. The implementation of `RLSpec` might accordingly be chosen as the CCS process $\mathtt{RLImp} := \nu x.\overline{\mathtt{up}}.(x + \overline{\mathtt{fail}}.\overline{\mathtt{down}}.x)$. We now establish that `RLImp` indeed refines `RLSpec` in the framework of our must–precongruence. First of all, it is easy to see by our characterization of $\sqsubseteq$ (cf. Thm. 3.2) that $\mathtt{ULSpec} \sqsubseteq \mathtt{RLImp}$, due to the internal *nondeterministic* choice in `ULSpec`. Further, we obviously have $\mathtt{RLImp} \models \mathtt{RG}$. Hence, we may infer by Thm. 3.5 that $\mathtt{RG} \sqsubseteq \mathtt{RLImp}$. Because `RLImp` cannot engage in an initial τ–transition, we may in summary conclude $\mathtt{ULSpec} \sqsubseteq \mathtt{RLImp}$ and $\mathtt{RG} \sqsubseteq \mathtt{RLImp}$. By Prop. 2.2, which is also valid for $\sqsubseteq$, and by Thm. 3.7, we derive $\mathtt{RLSpec} \equiv \mathtt{ULSpec} \wedge \mathtt{RG} \sqsubseteq \mathtt{RLImp} \wedge \mathtt{RLImp} \sqsubseteq \mathtt{RLImp}$, as desired.

When replacing in BLSpec the components RLSpec by RLImp, we obtain an implementation of our reliable bidirectional link, to which we refer as BLImp. Since $\sqsubseteq$ is a precongruence and $\mathtt{RLSpec} \sqsubseteq \mathtt{RLImp}$, we obtain $\mathtt{BLSpec} \sqsubseteq \mathtt{BLImp}$, i.e., BLImp refines BLSpec, which coincides with our intuition.

Finally, it is worth mentioning that LPC may actually be seen as a temporal logic that allows for some restricted form of branching–time reasoning. For example, the LPC process $\mathtt{sync} \implies (\overline{\mathtt{down1}}.\mathsf{tt} + \overline{\mathtt{down2}}.\mathsf{tt})$ encodes the property that the system state reached when executing action sync has both actions $\overline{\mathtt{down1}}$ and $\overline{\mathtt{down2}}$ enabled. Observe that, in contrast to $\overline{\mathtt{down1}}.\mathsf{tt} + \overline{\mathtt{down2}}.\mathsf{tt}$, the term $\overline{\mathtt{down1}}.\mathsf{tt} \wedge \overline{\mathtt{down2}}.\mathsf{tt}$ in LPC specifies the obvious contradiction that every initial transition is labeled by both actions $\overline{\mathtt{down1}}$ and $\overline{\mathtt{down2}}$ at the same time.

6 Conclusions and Future Work

We presented a novel *logical process calculus* LPC that integrates both classical process calculi, such as Milner's CCS, and temporal logics, such as the alternation–free linear–time μ–calculus $\mathsf{LT}\mu$. The syntax of LPC enriched CCS by operators for synchronous parallel composition (conjunction) and nondeterministic choice (disjunction), as well as by minimal fixed–points operators (finite unwindings of recursion). The semantics of LPC was given in terms of labeled transition systems and an unimplementability predicate, which are both defined via structural operational rules. A refinement preorder on process terms was then introduced, which conservatively extends both De Nicola's and Hennessy's must–preorder and the $\mathsf{LT}\mu$ satisfaction relation. Hence, $\mathsf{LT}\mu$ model checking may as well be understood as refinement checking. Finally, our must–preorder was shown to be compositional for all operators in LPC.

The outcome of our studies is a heterogeneous specification language, which allows system designers to specify systems in a mixed operational and declarative style, together with a behavioral preorder that permits component–based refinement. We believe that our setting provides groundwork for formally investigating those software engineering languages that support heterogeneous specifications as a mixture of operational state machines and declarative constraints, such as the *Unified Modeling Language* [4].

Regarding future work, we intend to study axiomatizations of our must–preorder and to develop an algorithm for its implementation in automated verification tools, such as the Concurrency Workbench NC [11]. It should also be investigated whether our approach is suitable for calculi other than CCS, such as Petri nets or the π–calculus, as well as for temporal logics other than $\mathsf{LT}\mu$, in particular for branching–time temporal logics.

Acknowledgments. We would like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] M. Abadi and L. Lamport. Composing specifications. *TOPLAS*, 15(1):73–132, 1993. See also: Conjoining Specifications, *TOPLAS*, 17(3):507–534, 1995.

[2] J.C.M. Baeten and J.A. Bergstra. Process algebra with a zero object. In *CONCUR' 90*, vol. 458 of *LNCS*, pp. 83–98, 1990.

[3] J.A. Bergstra, A. Ponse, and S.A. Smolka. *Handbook of Process Algebra*. Elsevier Science, 2001.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, 1998.

[5] J. Bradfield and C. Stirling. Modal logics and mu-calculi: An introduction. In *Handbook of Process Algebra*, pp. 293–330. Elsevier Science, 2001.

[6] E. Brinksma, A. Rensink, and W. Vogler. Fair testing. In *CONCUR '95*, vol. 962 of *LNCS*, pp. 313–328, 1995.

[7] S.D. Brookes and A.W. Roscoe. An improved failures model for communicating processes. In *Seminar on Concurrency*, vol. 197 of *LNCS*, pp. 281–305, 1984.

[8] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *LICS '89*, pp. 353–362. IEEE, 1989.

[9] R. Cleaveland and G. Lüttgen. Model checking is refinement: Relating Büchi testing and linear-time temporal logic. Techn. Rep. 2000-14, Institute for Computer Applications in Science and Engineering, March 2000.

[10] R. Cleaveland and G. Lüttgen. A semantic theory for heterogeneous system design. In *FSTTCS 2000*, vol. 1974 of *LNCS*, pp. 312–324, 2000.

[11] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *CAV '96*, vol. 1102 of *LNCS*, pp. 394–397, 1996.

[12] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *TCS*, 34:83–133, 1983.

[13] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *J. of the ACM*, 42(2):458–487, 1995.

[14] S. Graf and J. Sifakis. A logic for the description of non-deterministic programs and their properties. *Information and Control*, 68(1–3):254–270, 1986.

[15] O. Grumberg and D.E. Long. Model checking and modular verification. *TOPLAS*, 16(3):843–871, 1994.

[16] C. Hartonas. A fixpoint approach to finite delay and fairness. *TCS*, 198(1–2):131–158, 1998.

[17] M.C.B. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[18] M.C.B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. of the ACM*, 32(1):137–161, 1985.

[19] T. Hildebrandt. A fully abstract presheaf semantics of SCCS with finite delay. In *CTCS '99*, vol. 29 of *ENTCS*. Elsevier Science, 1999.

[20] R. Kaivola and A. Valmari. The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In *CONCUR '92*, vol. 630 of *LNCS*, pp. 207–221, 1992.

[21] O. Kupferman and M.Y. Vardi. Modular model checking. In *Compositionality: The Significant Difference*, vol. 1536 of *LNCS*, 1997.

[22] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton Univ. Press, 1994.

[23] L. Lamport. The temporal logic of actions. *TOPLAS*, 16(3):872–923, 1994.

[24] K.G. Larsen. The expressive power of implicit specifications. *TCS*, 114(1):119–147, 1993.

[25] M.G. Main. Trace, failure and testing equivalences for communicating processes. *J. of Par. Comp.*, 16(5):383–400, 1987.

[26] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[27] V. Natarajan and R. Cleaveland. Divergence and fair testing. In *ICALP '95*, vol. 944 of *LNCS*, pp. 684–695, 1995.

[28] A. Pnueli. Temporal logic of programs. In *FOCS '77*, pp. 46–57. IEEE, 1977.

[29] A. Puhakka and A. Valmari. Weakest-congruence results for livelock-preserving equivalences. In *CONCUR '99*, vol. 1664 of *LNCS*, pp. 510–524, 1999.

[30] A. Puhakka and A. Valmari. Liveness and fairness in process-algebraic verification. In *CONCUR 2001*, vol. 2154 of *LNCS*, pp. 202–217, 2001.

[31] B. Steffen and A. Ingólfsdóttir. Characteristic formulae for CCS with divergence. *Inform. & Comp.*, 110(1):149–163, 1994.

[32] C. Stirling. Modal logics for communicating systems. *TCS*, 49:311–347, 1987.

[33] C. Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science*, vol. 2, pp. 477–563. Oxford Univ. Press, 1992.

[34] A. Valmari and M. Tiernari. Compositional failure-based semantics models for basic LOTOS. *FAC*, 7(4):440–468, 1995.

[35] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS '86*, pp. 332–344. IEEE, 1986.